COPPE
UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

WEBTRANSACT: UMA INFRAESTRUTURA PARA ESPECIFICAÇÃO E COORDENAÇÃO DE COMPOSIÇÕES DE SERVIÇOS WEB

(WEBTRANSACT: A FRAMEWORK FOR SPECIFYING AND COORDINATING RELIABLE WEB SERVICES COMPOSITIONS)

RELATÓRIO TÉCNICO ES-578/02

Autores:

Paulo de Figueiredo Pires
Mário Roberto Folhadela Benevides
Marta Lima de Queirós Mattoso

RIO DE JANEIRO, RJ - BRASIL
ABRIL 2002

COPPE
FEDERAL UNIVERSITY OF RIO DE JANEIRO

WEBTRANSACT: A FRAMEWORK FOR SPECIFYING AND COORDINATING RELIABLE WEB SERVICES COMPOSITIONS

TECHNICAL REPORT ES-578/02

This work is an extended version of the thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of

Doctor of Science (D.Sc.)

in Computer Science and Systems Engineering

by

Paulo de Figueiredo Pires

Committee Members:

Prof. Marta Lima de Queirós Mattoso, D.Sc. (Chair, Advisor)
Prof. Mário Roberto Folhadela Benevides, Ph.D. (Advisor)
Prof. Asterio Kiyoshi Tanaka, Ph.D.
Prof. Louiqa Raschid, Ph.D.
Prof. Marco Antônio Casanova, Ph.D.
Prof. Valmir Carneiro Barbosa, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
APRIL 2002

ABSTRACT

WEBTRANSACT: A FRAMEWORK FOR SPECIFYING AND COORDINATING RELIABLE WEB SERVICES COMPOSITIONS

Paulo de Figueiredo Pires
April/2002

Advisors: Prof. Marta Lima de Queirós Mattoso
Prof. Mário Roberto Folhadela Benevides

Department: Systems Engineering and Computer Science

The recent evolution of internet technologies, mainly guided by the Extensible Markup Language (XML) and its related technologies, are extending the role of the World Wide Web from information interaction to service interaction. This next wave of the internet era is being driven by a concept named *Web services*. The Web services technology provides the underpinning to a new business opportunity, i.e., the possibility of providing value-added services through the composition of basic Web services. However, at its present stage, the Web services technology does not provide the necessary tools to build Web service compositions. The current Web services technology provides the necessary communication model for enabling message exchanges in the Web services environment. However, communication interoperability is only part of the problem when considering the building of reliable Web services compositions. Besides communication interoperability, the task of building Web service compositions requires mechanisms to deal with the inherent *autonomy*, and *heterogeneity* of Web services. Unlike components of traditional business process, Web services are typically provided by different organizations and they were designed not to be dependent of any collective computing entity. Since each organization has its own business rules, Web services must be treated as strictly autonomous units. Heterogeneity manifests itself through structural and semantic differences that may occur between semantically equivalent Web services. In a Web service environment it is likely to be found many different Web services offering the same semantic functionality thus, the task of building compositions has to deal somehow with this problem. Moreover, due to the autonomy and independency of Web service providers, it is not desirable nor even possible to rely on Web service providers to support transaction functionality. Thus, a transaction model for coordinating compositions cannot rely on Web service providers to support transaction facilities, such as the two-phase commit interface. Instead, it must be able to exploit the specific transaction support offered by each Web service provider (which can even be absent). The existence of semantically related Web services also influences the way transactions are coordinated. Since the same semantic service may be supported by many different Web services, the transaction model must exploit this feature to enhance the reliability of compositions.

In this work, we present a framework, named *WebTransact*, which encompasses all the requirements just described, thus providing the necessary infrastructure for building reliable Web service compositions. The WebTransact framework is composed of a multilayered architecture, an XML-based language, and a transaction model. The multilayered architecture of WebTransact separates the task of aggregating and homogenizing heterogeneous Web services from the task of specifying transaction interaction patterns, thus providing a new general mechanism to deal with the complexity introduced by a large number of Web services. The XML-based language, named *Web Service Transaction Language* (WSTL), is used for describing the transaction support and the content of Web services, for defining the aggregation rules of Web services, and for specifying the transaction interaction patterns of compositions. WSTL is an extension of WSDL (Web Services Description Language) thus it is adherent to the XML-based standards that enable Web service technology. The transaction model provides the adequate correctness guarantees when executing Web services compositions built with WSTL. The transaction model of WebTransact uses a new correctness criterion, named *2L-guaranteed-termination*, which is a weaker notion of atomicity that considers the needs of Web service environments. Still, we have developed formal definitions for reasoning on *safe* and *correct* execution of Web services compositions built with the mechanisms provided by the WebTransact framework. Use cases demonstrating the usage and potential of WebTransact are presented.

Index

FIGURES

1. Introduction

In this chapter, we present the motivation, objectives, and the scope of our work. Section 1.1 presents the evolution of Web technologies that enabled a new class of business process, named *Web services composition*. In Section 1.2, the novel issues that must be addressed when developing Web services compositions are pointed out. In Section 1.3, we define the goal and scope of our work and present how we address each one of the issues raised in the Section 1.2. Finally, Section 1.4 outlines the organization of this work.

1.1 MOTIVATION

During the Nineties, the World Wide Web was mostly used for interactive access to documents and applications. In almost all cases, such access has been done by human users, typically working through Web browsers. However, recent evolution of internet technologies, mainly guided by the Extensible Markup Language (XML) [138] and its related technologies, are extending the role of the World Wide Web from information interaction to service interaction. This next wave of the internet era is being driven by a concept named *Web services*.

Web services can be defined as modular programs, generally independent and self-describing, that can be discovered and invoked across the Internet or an enterprise intranet. Through Web services, one can encapsulate existing business processes, publish them as services, search for and subscribe to other services, and exchange information that cross the enterprise boundaries. Web services are the key technology to enable application-to-application interaction through the World Wide Web.

Web services combine aspects of component-based development ([63], [83], [109]) and the World Wide Web. Like components, Web services expose an interface that can be reused without worrying about how the service is implemented. Unlike current component technologies, Web services are not accessed via protocols dependent on a specific object-model, such as the Distributed Component Object Model (DCOM) [84], Remote Method Invocation (RMI) [126], or Internet Inter-ORB Protocol (IIOP) [97]. Instead, Web services are accessed via ubiquitous Web protocols and data formats, such

as Hypertext Transfer Protocol (HTTP) and XML, which are vendor independent. This is a key difference between Web services and the current technologies that implement component-based development. Due to the use of ubiquitous Web protocols and data formats, Web services achieve a higher degree of interoperability with respect to the current technologies. The current Web services technology can be considered as an open client-server architecture that enables the interoperability between disparate systems without using any proprietary client libraries, thus simplifying the traditional development of client-server applications by effectively eliminating code dependencies between clients and servers.

Web services are, in fact, an evolution of the existent World Wide Web standards and protocols. The HTML standard [139] that displays Web pages content has evolved into a more generalized standard, XML [138], which can be used to describe not only Web pages content but to encode virtually any data. The HTTP protocol ([37], [95]) has been extended with *SOAP*[1] [144], which provides a more generalized communication format between two independent systems. *Web Services Description Language* (WSDL) [137] is an XML language for describing the interface of a Web service enabling a program to understand how it can interact with a Web service. Web services can be published in a directory to be found or discovered and used by other programs using a new specification named *Universal Description, Discovery, and Integration* (UDDI) [133], which has evolved from the *Lightweight Directory Access Protocol* (LDAP) [149].

Web services are typically built with XML, SOAP, WSDL, and UDDI specifications. Today, the majority of the software companies are implementing tools based on these new standards. To name a few, we have Microsoft's .Net platform ([88], [89]), the IBM Dynamic E-business platform ([58], [59]), Sun Open Net Environment (Sun ONE) ([123], [125]), WebLogic Integration 2.0 platform from BEA Systems [10], and IONA's Orbix E2A Web services integration platform [61]. Considering how fast implementations of these standards are becoming available, along with the strong commitment of several important software companies, we believe that they will soon be as widely implemented as HTML is today.

[1] Up to version 1.1, SOAP was the acronym of "Simple Object Access Protocol". However, SOAP is not an acronym anymore (since the release of version 1.2).

According to the scenario just described, an increasing number of on-line services will be published in the Web during the next years. As these services become available in the Web service environment, a new business opportunity is created, i.e., the possibility of providing *value-added* Web services. Such value-added Web services may be built through the integration and composition of Web services available on the Web [22].

Web services composition is the ability of one business to provide value-added services to its customers through the composition of basic Web services, possibly offered by different companies ([22], [23], [55]). Web services composition shares many requirements with business process management ([4], [49], [65], [116]). They both need to coordinate the sequence of service invocation within a composition, to manage the data flow between services, and to manage execution of compositions as transaction units. In addition, they need to provide high availability, reliability, and scalability. However, the task of building Web services compositions is much more difficult due to the degree of *autonomy*, *heterogeneity* and *dynamism* of Web services. Unlike components of traditional business process, Web services are typically provided by different organizations and they were designed not to be dependent on any collective computing entity. Since each organization has its own business rules, Web services must be treated as strictly autonomous units. Heterogeneity manifests itself through structural and semantic differences that may occur between semantically equivalent Web services. In a Web service environment it is likely to be found many different Web services offering the same semantic functionality thus, the task of building compositions has to somehow deal with this problem. Building Web services compositions is further complicated by the fact that the Web service environment is highly dynamic. The Web is a truly dynamic environment and Web services are not an exception. New Web services will by published at a fast rate and published Web services may become unavailable in the same rate. Moreover, Web services will not expose a static behavior, just because they must evolve according to the rules of its associated business market.

Web services composition provides the underpinning required to enable services-oriented architecture on the Web. However, the building of value-added services on this new environment is not a trivial task. Due to the many singularities of the Web service environment, it is not possible to rely on the current models and solutions to build and coordinate compositions of Web services.

1.2 ISSUES FOR DEVELOPING WEB SERVICES COMPOSITION

The WSDL deals with interoperability between processes on the Web providing a framework that serves as the basic infrastructure to build Web services compositions. However, only the peer-to-peer interaction between client programs and Web services is not sufficient to build reliable Web services compositions. Building reliable compositions requires mechanisms for coordinating transactions that span multiple and *autonomous* Web services, requiring a loosely coupled mechanism to link Web services to compositions that deals with the inherent *heterogeneity* and *dynamism* of Web service interfaces. Moreover, the task of building compositions can be significantly improved if the composition developer does not have to deal with such *heterogeneity* and *dynamism* of Web service interfaces. Therefore, a framework for developing Web services compositions must provide mechanisms to homogenize and isolate such interfaces before they are used to build compositions. So far, a framework addressing all these issues is presently missing from the Web services platform.

In the next sections, we present our vision of the necessary requirements for supporting transactions and homogenization of services in the context of Web services compositions.

1.2.1 Transaction Coordination in Web Service Context

Traditionally, transactions are expected to satisfy the properties of *atomicity*, *isolation*, *consistency*, and *durability* known as ACID properties ([47], [15], [78]). These properties are used to define the notion of *correctness* of traditional transaction models and protocols. In a Web service environment, supporting ACID properties is not desirable or even feasible.

The traditional notion of *atomicity* means that the steps executed during a transaction are executed completely or not at all. Such strict notion of atomicity is not reasonable for Web service environment. A composition must exploit the different transaction support provided by Web services as well the existence of many semantically equivalent Web services. For instance, if a Web service supports compensation then the composition must exploit it to deal with failures that may occur during its execution. The same must be done when executing services that are supported by many Web services that are semantically equivalent. Whenever a Web service fails, it must be verified whether

there is another Web service that supports the desired semantics. Therefore, a Web services composition may accomplish its objectives even if some of its steps (Web services) were not executed successfully.

Consistency means that a transaction takes the system from one consistent state to another. In order to guarantee consistency across multiple Web services, local subsystems must be monitored to identify service invocations that violate global dependencies [114]. Such monitoring is unreasonable in a Web service environment due to the strict autonomy and the potentially huge number of Web services.

Isolation is concerned with controlling the concurrent execution of transactions to provide the illusion that concurrent transactions are executed in a serial order. Generally, locking mechanisms are used to provide isolation. Since Web services may support dissimilar transaction support, the underlining system may not support or allow locking through Web service access. Moreover, isolation is also costly because transaction compositions may be long running, and providing isolation for long-running transactions would deteriorate the underlining system performance.

The only property that must be fully supported in transaction compositions is *durability*. The traditional notion of durability means that when a transaction completes its execution, all of its actions are made persistent, even in presence of system failures. In the same way, durability must be ensured in Web service environments.

According to the discussion in the previous paragraphs, a transaction model to coordinate composition executions must ensure durability as well as support a relaxed notion of atomicity that considers both the dissimilar transaction support and the existence of semantically equivalent Web services.

There are many transaction models that propose relaxing (some of) the ACID properties ([31], [110]). These transaction models, known as *Extended Transaction Models* (ETM), have been proposed to support specialized applications that require transaction models more flexible than the ACID-based models. Examples of ETM include: closed nested transaction [93], sagas [43], contracts [145], flexible transaction [33], and multitransaction [17]. However, ETMs do not fully address the requirements of transaction coordination of Web services compositions.

The problem with adopting the ETMs for coordinating the transactional execution of Web services compositions is three-fold. First, ETMs were not designed to deal with semantically equivalent, but syntactically and behaviorally dissimilar processing entities, which is likely to occur in the Web services environment. Second, ETMs have been designed with the assumption that all processing entities provide support for a set of transaction facilities. For example, ETMs for multidatabase [108] require that each processing entity expose a two-phase commit interface ([46], [72], [71]), restricting the processing entities to database systems. In a Web service environment, this functionality may not be present, since we are dealing with services running inside arbitrary systems that may not have transaction support or may not expose transactional interfaces. Third, all ETMs have been designed to support (some level of) the consistency property, which is not feasible in a Web service environment. Since ETMs are not suitable for coordinating transaction executions of Web services compositions, there is a need for a new transaction model tailored for this kind of environment.

To conclude this section, we summarize the issues on transaction coordination for Web service environments:

- **Transaction support of Web service providers**. Web service providers are independent and autonomous; it is not desirable nor even possible to rely on Web service providers to support transaction functionality. The transaction model cannot rely on Web service providers to support transaction facilities like a two-phase commit interface.

- **Dissimilar transaction support of Web services**. The transaction model must exploit the specific transaction support offered by each Web service provider.

- **Semantically equivalent Web services**. Since the same semantic service may be supported by many different Web services, the transaction model must exploit this feature to enhance the reliability of compositions.

1.2.2 Homogenization of Web Services

Another important aspect to be addressed in a composite Web services environment is the aggregation of semantically equivalent Web services. As the number of available Web services increases, it is likely to appear many different Web services, implemented

by different companies, providing the same semantic functionality. The process of aggregating Web services involves the resolution of *semantic dissimilarities* that may exist among that services and the description of its *content capabilities*. For instance, consider two Web services providing car reservation from two different companies. It is unlike that both services expect the same input message format. The dissimilarities that might occur when aggregating Web services can range from naming to structural dissimilarities. The problem of solving semantic dissimilarities of services is similar to the problem of solving data dissimilarities occurring in mediator systems ([19], [77], [42], [106], [132]) and heterogeneous database systems ([19], [32], [108]). The techniques employed to solve the data dissimilarities can be adapted to solve semantic dissimilarities. Some other works already address the problem of solving semantic dissimilarities in the Web through Wrapper mediator technology ([50], [135]). It is important to note that such techniques cannot be *directly* used to solve semantic dissimilarities of services. The problem of solving data dissimilarities is commonly related to the homogenization of different (data) schemas while the problem of solving semantic dissimilarities of Web services is related the homogenization of different service interfaces (message formats).

Besides semantic dissimilarities, Web services are different regarding its content capabilities. For instance, a car reservation service from company *A* might be able to make reservation only in Brazil, while a car reservation service form company *B* might be able to make world wide reservations. Such information is particularly important when executing a Web services composition. Whether the content capability of Web services is available, it is possible select only those Web services capable of handling that execution, avoiding unnecessary calls to other Web services.

Considering the number of companies currently connected to the World Wide Web, there is a real potential for appearing a huge number of Web services in the near future. Therefore, it is important that a system, willing to offer value-added services, provide mechanisms for aggregating those Web services. Web services aggregation facilitates the building of compositions. A business process can be composed of services built upon a layer of aggregated Web services. Such layer hides the dissimilarities, and even the existence of many different Web services, providing a single common interface, which is used to build the composition. Therefore, a developer of a composition can concentrate in the business rules while building a value-added service without having to deal with the specific behavior of each available Web service. Note that a composite Web service

environment may have a massive number of available Web services. Therefore, without mechanisms for organizing and homogenizing such environment, it would be very unlikely that one could take some benefit when composing value-added service.

1.3 GOAL AND SCOPE OF THIS WORK

The main goal of this work is the development and specification of a framework to build reliable Web services compositions. In order to achieve this goal we have developed *WebTransact*[2]. WebTransact is a framework that deals with the specific problems of building Web services compositions. Such framework is composed of a multilayered architecture, an XML-based language and a transaction model.

The *multilayered architecture* is composed of several specialized components that deal with the problem of homogenizing Web services. The WebTransact architecture employs the mediator wrapper technology. Wrapper mediator systems ([19], [77], [42], [106], [132]) have been successfully developed to mediate the *query capability* of remote (database or non-database) sources and to provide information integration at the mediator level. In WebTransact, these techniques were adapted to resolve *semantic dissimilarity* and to describe *content capability* of Web services.

The *XML-based* language, named *Web Service Transaction language* (WSTL), is used for describing the transaction support and content of Web services, for defining the aggregation rules of Web services, and for specifying the transaction interaction patterns of compositions. WSTL is an extension of WSDL thus it is adherent to the XML-based standards that enable Web service technology.

The *transaction model* provides the adequate correctness guarantees when executing Web services compositions built with WSTL.

Compositions are built over a layer of homogenized Web services. Therefore, the composition developer does not have to care about transaction or semantic dissimilarities or even the content capability of different Web services. Such heterogeneity is resolved when Web services are integrated to the WebTransact architecture.

[2] The fundamental ideas behind the WebTransact framework have their origin in the MedTransact architecture [107]. MedTransact was developed during the year 2000, when I was working with Professor Louiqa Raschid at University of Maryland.

A composition developer specifies composition through WSTL. The constructors provided by WSTL allow not only the specification of the control and data flow between the services of a composition but also allow the specification of different levels of transaction reliability and execution optimization. For example, a developer can explicitly define, through these constructors, the action to be done when a given call to a Web service fails. Additionally, the developer can specify a set of Web services to be performed in parallel and then, he can define rules to select those Web services that must be committed or compensated, after their execution.

The transaction model of WebTransact uses a correctness criterion, named *2L-guaranteed-termination*, which is a weaker notion of atomicity that considers the needs of Web service environments. Based on this criterion, the transaction model of WebTransact defines the protocols that guarantee the correct and safe execution of Web services compositions. Correct execution means the composition will be executed according to its specification, i.e., the transaction semantics specified by the user is ensured by the WebTransact, and that the *2L-guaranteed-termination* property is assured. Safe execution means the WebTransact only execute composition specifications whose Web services support the necessary transaction behavior, i.e., if a Web service might be compensated after its execution then it must support compensation. The notion of correctness and safeness employed by WebTransact is adapted from both Extended Transaction Models ([31], [110]) and transactional process coordination ([1], [3], [113], [114]).

The WebTransact framework is a multidisciplinary work that is related with many other areas such as e-service composition, transactional process coordination, workflow management systems, and distributed computing systems.

There has been extensive research in transaction support for distributed computing systems ([7], [19], [26], [31], [41], [44], [98]), transactional process coordination ([2], [3], [113], [114]), and in workflow management systems ([29], [45], [56], [79], [90], [112], [145]). While these projects address the support of distributed transactions, they do not consider the coordination of service capabilities of autonomous remote service providers. Since Web services support dissimilar capabilities with respect to its transaction behavior, this is a significant difference. Thus, implementing transaction semantics across the Web services becomes much more difficult, compared to a scenario where all distributed components support identical transaction behavior. Moreover, these works were

conceived before the current stage of the World Wide Web. Therefore, they do not consider the XML-based standards that enable the Web service technology.

More recently, the area of e-service composition is attracting the interest of both academia ([22], [55]) and industry ([8], [23], [76], [129], [136]). The existent work in this area is concentrated in defining primitives for composing services and automating service coordination. The majority of these works consider XML-based standards for Web service technology. However, the proposed primitives for composition do not directly address the problems associated with the necessary *homogenization* of Web services. Still, the transaction support proposed in this area does not consider the coordination and mediation of Web services with dissimilar transaction support.

To the best of our knowledge, there are no other works on integrated frameworks that address the problem of building reliable Web services compositions. WebTransact addresses this new class of transaction interactions involving Web services from multiple organizations on the Web through mediating dissimilar service capabilities of Web services, while supporting distributed transaction semantics across the mediator and the Web service providers.

1.4 ORGANIZATION

The remainder of this work is organized as follows.

In Chapter 2, we present a more accurate view of the requirements for developing Web services compositions and we describe how WebTransact addresses these requirements. In the discussion on that requirements, a survey on the existent distributed transaction models is presented and, based on such models, a new model, suited for Web services environments, is proposed. Next, the technologies and models that are strongly related with WebTransact are described. We conclude this chapter with a brief overview on the Web Service Description Language (WSDL).

In Chapter 3, we present an overview of the WebTransact framework. The goal of this chapter is to provide the reader a unified view of all components of the WebTransact framework. First, we describe the WebTransact architecture. Next, we sketch the main components of such architecture. First, we describe the remote service, which is the component responsible for integrating Web services. Then, we present the mediator

service, the component responsible for homogenizing Web services. Next, we describe how Web services compositions are built using a layer of homogenized Web services. Finally, we briefly describe the WebTransact execution model.

In Chapter 4, we show how to describe the dissimilar transaction behavior of Web services. This chapter starts by introducing the transaction language of WebTransact, WSTL, and by defining the relationship between WSTL and WSDL. Next, we present how to expose the dissimilar transaction support of Web services. This is done through the explanation of the WSTL elements that define the different types of transaction support of Web services.

In Chapter 5, we describe how Web services are integrated in the WebTransact framework. In this chapter, we first define the *mediator service*, the component of the WebTransact framework for specifying the homogenized interface of semantically equivalent Web services. Then, we define the *remote service*, the component of the WebTransact framework responsible for resolving the semantic and content dissimilarities of Web services.

In Chapter 6, we present our reference model for specifying compositions. First, we present the abstract concepts of the reference model. Then, we explain these concepts throughout an example. Next, we present the WSTL constructors used to specify Web services compositions. The last part of this chapter is dedicated to the detailed description of the WSTL elements used to specify compositions.

In Chapter 7, we describe the execution model of WebTransact. This chapter has two inter-related parts. The first part presents transaction interactions occurring at the execution level of WebTransact compositions. The second part presents transaction interactions occurring at the execution level of a particular mediator service operation. In this chapter, the algorithms that coordinate such interactions are specified. In addition, a formal definition for reasoning on composition specifications and executions are presented.

Finally, Chapter 8 presents the concluding remarks and the future works.

2. Transactional Composition of Web Services

In this chapter, we present the requirements of developing Web services compositions showing why such development is different from current distributed computing. One of such requirements, the transaction support, is especially important for our work. Therefore, after the requirements exposition, we present an overview of the current transaction processing technologies. Then, we make an analysis of the applicability of those current transaction processing technologies to the Web services environment. Such analysis demonstrates the inadequacy of current transaction processing technologies and point out the issues that must be addressed for the development of a transaction model suited for the Web services environment. Next, we present a brief review of the available technologies that can be used as infrastructure for building Web services compositions. All the presented technologies influence the solutions adopted in the development of WebTransact. We conclude this chapter with an overview of the current Web service technology, named Web Service Description Language (WSDL) [137] and SOAP [144]. This overview is concluded with an analysis of the features that are lacking in WSDL and SOAP when considering the development of Web services compositions.

2.1 REQUIREMENTS FOR DEVELOPING WEB SERVICES COMPOSITIONS

The basic idea behind the Web service concept serves the same purpose that CORBA [97], COM [84], and EJB [124] serve for current distributed computing, i.e., enabling interoperability among system components. However, there is an important distinction between the infrastructure for Web services as opposed to traditional infrastructures. The interoperability problem among Web services is more challenging than the interoperability between current distributed computing. One reason for this is that Web services may be separated by firewalls and the semantics of data being communicated between them may not be uniform at the communicating ends. Another important distinction between Web services and traditional distributed component models is that Web services can be developed, published and maintained by completely independent companies. That means that Web services need not only describe its interface and message format but also its *behavior*. For example, a Web service must provide a

means to answer questions like: *The Web service can participate in a distributed transaction? If so, which transaction models and protocols are supported? Which type of content, the service is able to handle? How much is charged to use the service? Is it possible to pay for a better quality of service?* The point is: the potential utilization of a Web service is directly related to its ability to describe itself. A standard description of the behavioral properties exposes the interaction patterns that are supported by Web services allowing them to participate in different kinds of compositions.

The high degree of autonomy and independency of Web services creates a huge variety of interaction patterns that makes the task of composing such services into a value-added service much more difficult than it is in the current distributed computing.

Having pointed out that composing Web services has new challenges regarding current distributed computing, we present the issues to be addressed when developing value-added services on top of a Web service environment:

Enabling standards: The Web service infrastructure is built upon XML-based standards, such as XML [138], WSDL [137], SOAP [144], and UDDI [133] (Figure 2.1). These standards describe the Web service interface and the supported communication pattern. A framework for composing Web services must be adherent to these standards.

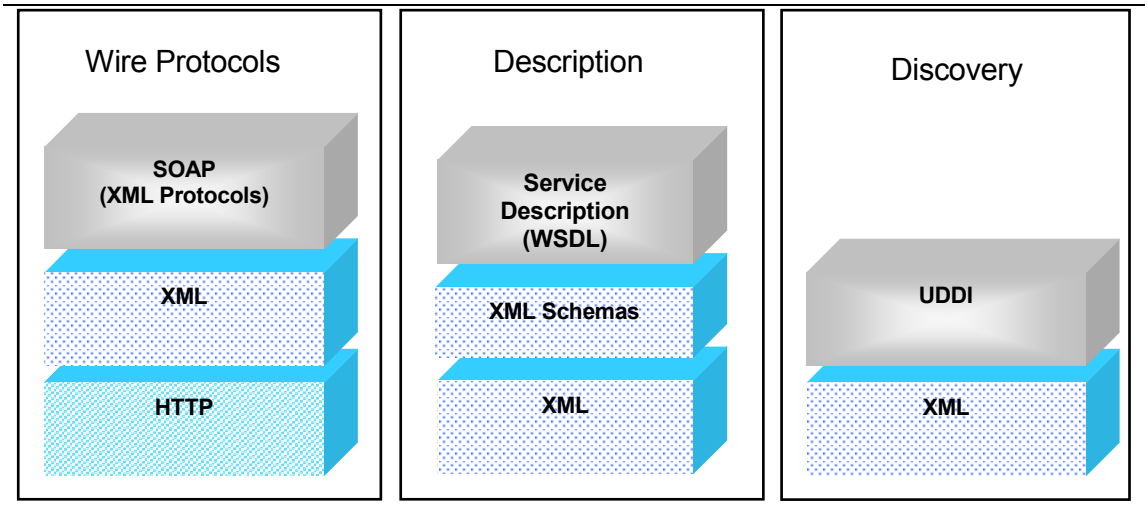


Figure 2.1 - XML-based technologies for enabling Web services.

Autonomy: A framework for composing Web services must minimize the impact of changes to a Web service. The Web services should follow the evolution of its related

business rules, thus changes are likely to occur in both interface and behavior of Web services. Changes in Web services must not affect existent Web services compositions.

Binding Mechanisms: Typically, application designers bind software components to one another at development time. Since Web services are likely to be implemented and provided by different service providers, a composition framework must be able to provide flexible binding mechanisms.

Semantic dissimilarities. The meaning and structure of the data communicated among companies is dependent on each company interpretation. For instance, the room price field of a hotel reservation may have different significance and structure for different hotels. The process of composing Web services must consider such semantics dissimilarities providing mechanisms to solve it. The task of composing value-added services on top of such heterogeneous Web services can be a very complex task. However, such complexity can be minimized if a *mediation layer* is introduced between the Web services and the composition. The mediation layer can act as a homogenizing layer regarding the semantic dissimilarities of Web services. Semantically equivalent Web services can be aggregated in a new abstract service. Such service can expose a homogenized interface of those Web services and then, service compositions can be built on top of these abstract services.

Content capability. Besides semantics dissimilarities, Web services are different regarding its *content capabilities*. For instance, a car reservation service from company *A* might be able to make reservation only in Brazil, while a car reservation service from company *B* might be able to make worldwide reservations. Such information is particularly important when executing a Web services composition. When the capability of Web services is available, it is possible select only the adequate Web services for that execution, avoiding unnecessary calls to other Web services.

Transaction support. Whereas Web service providers are independent and autonomous, they can support dissimilar transaction functionalities. Therefore, Web services should make their transactional properties publicly available. Transactional properties should be part of the Web service interface rather than a hidden aspect of its backend. The transactional behavior of a service can then be exploited by other services to simplify their error-handling logic and to make entire compositions interactions

transactional. However, such transactions interactions are challenging to implement because they span multiple companies and because the underlying transaction protocols execute over wide-area networks. Different companies can expose Web services with different and dissimilar transaction support thus coordinating a composition execution means coordinating transactions with dissimilar support and through different transaction interfaces. Communication over wide-area networks is generally done throughout firewalls thus constraining the communication models and protocols that can be used to coordinate transactions.

A framework for enabling Web services compositions must addresses all the requirements just described. When considering reliable compositions, the requirement of transaction support is of special interest. Therefore, in the next sections we present an extended discussion on the role of transactions in the Web service environment. First, we review the basic concepts related to the current transaction processing technologies (Section 2.2). Next, we present an analysis on the applicability of such technologies to the Web service environment. This analysis demonstrates that the current transaction processing technologies cannot be directly applied to support transaction in the Web services environment.

2.2 TRANSACTION PROCESSING

Transaction processing is the software technology that makes distributed computing reliable ([15], [21]). Transactions are said to provide the ACID properties ([15], [47]):

- **Atomicity**. A transaction executes completely or not at all, i.e., a transaction either commits or aborts. If a transaction commits, all of its effects are made persistent. If it aborts, all of its effects are undone.

- **Consistency**. A transaction is a correct transformation of the system state thus preserving its internal state consistency.

- **Isolation**. Concurrent transactions are isolated from the updates of other incomplete transactions.

- **Durability**. Once a transaction commits, its effects will persist even if there are system failures.

The ACID properties provide the illusion that each transaction execute as if no other transaction were executing concurrently and as if there were no failures. Therefore, the program developers do not have to deal with the complexity of handling concurrency and failure code, allowing them to focus on designing the application.

Two different sets of protocols usually ensure the ACID properties [1]: the *concurrency control* protocols and the *recovery protocols*. The concurrency control protocols ensure the *execution atomicity* while the recovery protocols ensure the *failure atomicity*. The execution atomicity refers to the problem of ensuring the overall consistency of the database, even in the presence of concurrently executing transactions. The failure atomicity ensures the atomicity as well as isolation and durability properties of transactions.

Traditionally, the execution atomicity is ensured by protocols that are built on the concept of *serializability* [14]. This concept defines the main notion of correctness for the execution of a set of transactions. The key idea behind serializability is that since serial executions of a set of transactions are correct, then any execution whose behavior is equivalent of some serial execution must also be correct. In most of the concurrency control protocols, serializability is based on the notion of conflicting operations and is called *conflict serializability*. Different versions of serializability, such as *view* and *final-state* serializability [99], have different notions of equivalence. For example, in two equivalent executions, view serializability requires that each transaction reads the same values and that the final value of the data is the same and produced by the same transactions. Concurrency control protocols, such as two-phase locking [34] and timestamp ordering [13], are used to ensure the serializable execution of transactions.

Failure atomicity is ensured by recovery schemes that deal with both *transaction failures*, such as a transaction abort due to negative balance in a money transfer operation, and *systems failures*, such as a hard disk crash. In order to handle transaction failures in the presence of concurrency, the concurrency control algorithms have to ensure not only serializability but also *recoverability*. A common way of ensuring recoverability is to enforce a transaction to only read and write data values written by committed transactions (a concept known as *strictness*) [52]. For example, concurrency control protocols based on locks can be extended to guarantee that a transaction holds its write locks until commit time. This leads to a modified version of the two-phase locking protocol known as *strict*

two-phase locking. In concurrency control protocols that ensure strict executions, an aborted transaction is undone by simply restoring the old data values, called *before-images*, corresponding to the write operations executed by the aborted transaction. On the other hand, system failures are dealt through two different processes: *redo process* and *undo process*. The redo process deals with committed transactions that were not incorporated into the database when the failure occurred, these transactions are written, by the redo process, to the database during a failure recovery procedure. The undo process deals with transactions that were active or aborted at the time the failure occurred, for each one of these transactions, the write operations that were incorporated into the database are undone during the recovery procedure. The needed information for the redo and the undo process are kept in the *system log* on stable storage, so that it survives system failures.

In homogeneous distributed systems, i.e., distributed databases, the execution atomicity of transactions can be ensured by extending the centralized protocols for concurrency control. For example, distributed two-phase locking can be implemented by requiring a transaction to release a lock only after it has obtained all its locks on all necessary local sites. This constraint can be easily enforced by using the strict two-phase locking protocol. On the other hand, ensuring failure atomicity is more complex than in centralized systems. Several protocols ([46], [70], [92]) have been proposed to solve the distributed commitment problem, out of which the most widely used is the two-phase commit protocol (2PC) ([46], [71], [72]). In the next section, we review the 2PC protocol as well as the current architectures that support such protocol.

2.2.1 Distributed Transaction Processing

The two-phase commit protocol (2PC) ([46], [71], [72]) is the key technology required to support distributed transaction. In the 2PC protocol, the distributed transaction is executed in two message phases, under the supervision of a coordinator. In the first phase, the coordinator sends a message to each of the participating nodes asking whether it agrees to commit the transaction. If the participant detects no local problem on that transaction, it agrees to locally commit the transaction, returning a yes vote to the coordinator. Otherwise, it votes no. When a participant votes yes, it enters into a so-called *in-doubt* state where it has no right to abort the transaction unilaterally. After receiving all votes, the coordinator starts the second phase of the protocol. If all participants vote yes,

then the coordinator will decide on global commit. If at least one participant replies with a no vote or a timeout is reached from any participant, then the coordinator will decide on global abort. The second phase ends when the coordinator notifies all in-doubt participants on the transaction commit or abort. Along these events, both the coordinator and the participants must record logging information to survive crashes.

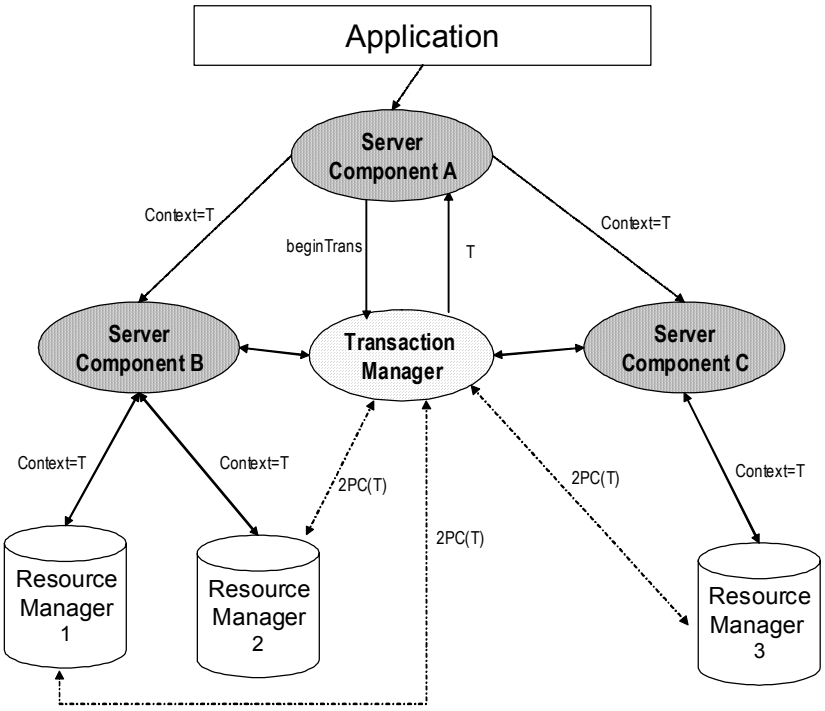


Figure 2.2 - Distributed Transaction Architecture.

A distributed transaction architecture that supports the two-phase commit protocol is shown in Figure 2.2. Note that, although there are a number of possible architectures, the one shown in Figure 2.2 represents the fundamental features and components. *Server components* are programs that implement the business transactions and act as clients for the transactional resources. *Resource managers* are components that manage persistent data repositories, and are coordinated by a central *transaction manager*. The transaction manager is responsible for generating *transaction identifiers* and coordinating the two-phase commit protocol. Server components can invoke each other and directly access their local data repositories throughout resource managers on behalf of some transaction with transaction identifier `T`. When a server component calls a local resource manager `RM` for the first time on behalf of transaction `T`, `RM` calls its local transaction manager asking for enlisting `RM` in `T`. This process is called *enlistment* and it allows the transaction manager to keep track of all resources involved in a given transaction. When the

transaction manager receives a commit or abort message for T, it begins the two-phase commit protocol with all the local resource managers enlisted in the transaction, using the transaction identifier T.

2.2.2 Distributed Transaction Coordination

In the architecture described in the previous section, the transaction coordination is centralized in the transaction manager: it generates the transaction identifier; it has the knowledge on all resources involved in the transaction; and it runs the two-phase commit protocol. Such centralized architecture is not realistic when large-scale distributed environments are considered, as is the case of systems based on Web services [35]. In a Web services environment, multiple information systems should interact through transactions. Thus, more than one transaction manager will be involved in the same distributed transaction. Figure 2.3 shows an example of distributed transactional system with multiple transaction managers. In this architecture, a transaction manager is the coordinator of the transaction, its participants are local resource managers accessed by the transaction, and remote transaction managers at nodes where the transaction executes. Therefore, each transaction manager can be the coordinator or a participant of a given transaction. In the same way, resource managers must enlist in a transaction when there is only one transaction manager. In this architecture, a transaction manager that plays the participant role must enlist in a transaction when it is first accessed on behalf of that transaction. The enlistment process for local calls is the same as described for a centralized transaction manager. For remote calls, when a server component SC_A at a local site A calls another server component SC_B at a remote site B on behalf of transaction T_A for the first time, both SC_A and SC_B transaction managers must be notified that transaction T_A is crossing the boundaries of site A towards site B. This remote communication process is typically done by a component called *communication manager*.

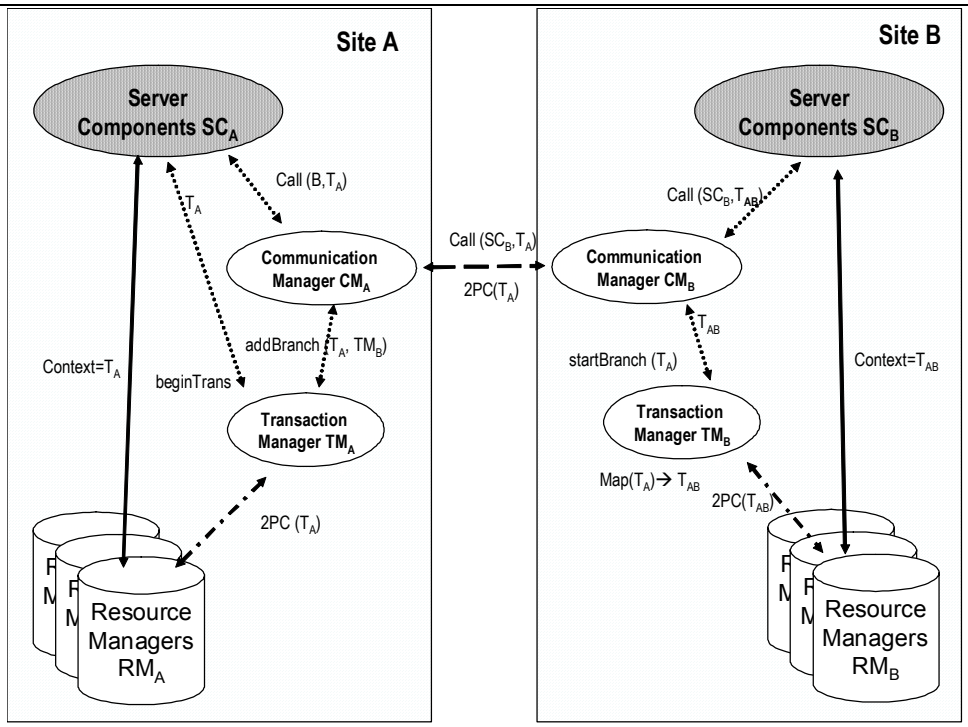


Figure 2.3 - Distributed transaction architecture with multiple transaction managers.

For example, in Figure 2.3, server component SC_A running the transaction T_A at site A calls server component SC_B at site B. This call is made throughout communication managers CM_A and CM_B. When CM_A receives such call, it informs the local transaction manager TM_A that transaction manager TM_B of site B is also running T_A. This is necessary so that transaction manager TM_A knows that transaction manager TM_B must be included in the two-phase commit coordination. At site B, communication manager CM_B is responsible for calling server component SC_B on behalf of transaction T_A and for informing the transaction manager TM_B that it is playing the role of participant in executing T_A, which is being coordinated by a transaction manager TM_A. From this point on, the transaction manager TM_B is ready to receive enlist messages from its local resource managers RM_B and two-phase commit messages from transaction manager TM_A. Because each transaction monitor works with its own policies for determining a transaction identifier, a transaction that spans multiple transaction managers must have multiple and possibly different identifiers in each system. Therefore, it is necessary to perform a transaction identifier mapping from one system to another when invocations cross transaction manager boundaries. There are two possible models for doing this mapping: the *push model* and the *pull model* [47].

To explain these models consider that a server component SC_A has created a transaction T_A and needs to invoke another server component SC_B at site B on behalf of transaction T_A.

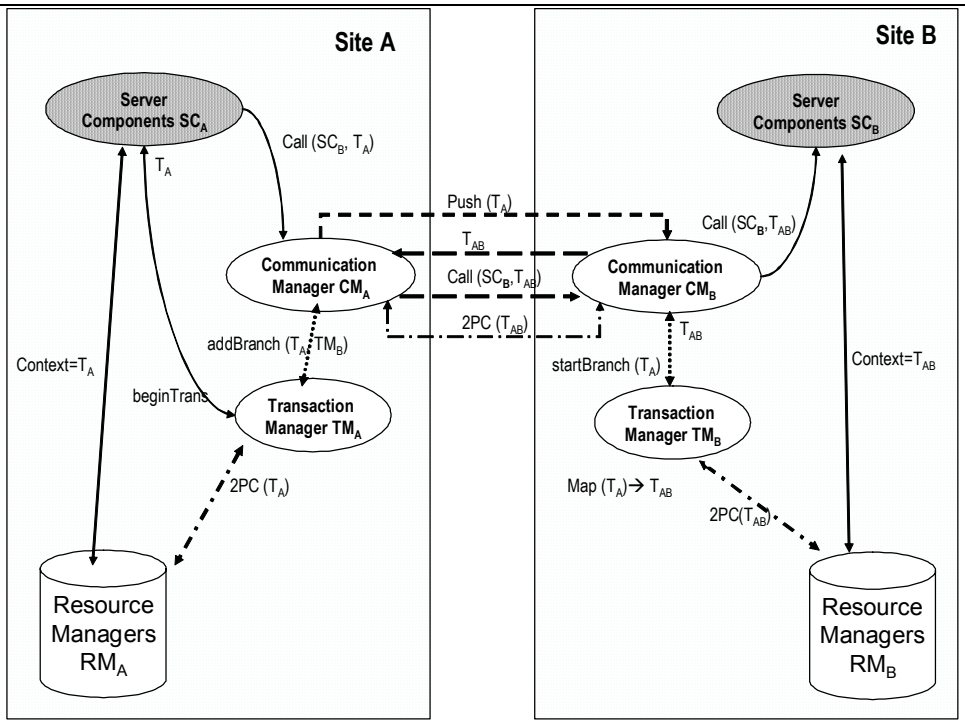


Figure 2.4 - The Push Model.

In the *push model* [47] (Figure 2.4), the server component SC_A first asks his transaction manager TM_A to export transaction T_A to SC_B`s transaction manager at B. Then, transaction manager TM_A sends a message to transaction manager TM_B asking it to instantiate transaction T_A as a subordinate of transaction manager TM_A, and returns its local transaction identifier, say T_{AB}, for transaction T_A. The server component SC_A then invokes server component SC_B on site B passing the transaction context T_{AB} along the invocation. At site B, this invocation will run as local transaction, T_{AB}. When TM_A sends transaction T_A to TM_B, it enlists TM_B as a participant in T_A (`addBranch` operation in Figure 2.4). Thus, TM_A knows it must involve TM_B in the two-phase commit protocol for transaction T_A.

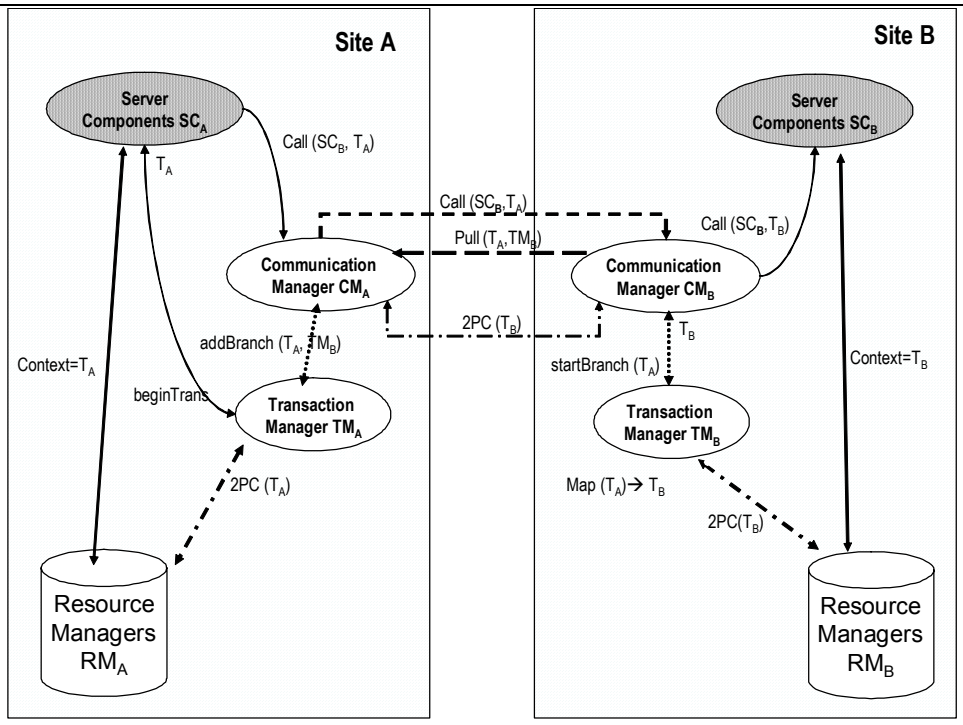


Figure 2.5 - The Pull Model.

In the *pull model* [47] (Figure 2.5), the server component SC_A invokes server component SC_B directly, passing the transaction identifier T_A along the invocation. The server component SC_B asks its transaction manager TM_B to enlist it in the transaction T_A. When transaction manager TM_B receives this enlist message, it will *pull* T_A from transaction manager TM_A. Because of this operation, transaction manager TM_A knows it must involve transaction manager TM_B in the two-phase commit protocol. Note that all communication goes through each site's communication manager.

2.3 INADEQUACY OF CURRENT TRANSACTION PROCESSING FOR THE WEB

The transaction support requirements of the Web services environment make the task of coordinating transactions in such environment more challenging than in traditional distributed transaction systems. Due to the inherent autonomy and dissimilar capabilities of Web service, it is not possible to rely on such services to support transaction functionality such as the 2PC protocol. To preserve autonomy, most of the Web services are likely to be designed to behave as a single autonomous transaction, i.e., each invocation to the Web service will produce a well-defined outcome: an invocation can produce a valid result and, therefore, it has committed or it can send back an error and, therefore, it has aborted. In this way, a transaction involving multiple Web services is

composed of many autonomous subtransactions that abort or commit independently. Therefore, the commitment of such transactions cannot be guaranteed by transaction commitment protocols like 2PC. If each Web service invocation is executed as an independent transaction, the only way of guaranteeing the desired all-or-nothing property of transactions is through the notion of *compensation*. The effects of a Web service execution can be compensated by executing another Web service that semantically undoes the effects of that Web service execution. However, due to their dissimilar capabilities, not all Web services will support compensation and those that support must somehow expose this behavior. The transaction issues just described make the task of coordinating transactions that spam multiple Web services, as is the case of Web services compositions, much more complex than it is in the current distributed transaction systems.

In the next sections, we present an analysis of the limitations of the current transaction processing technologies regarding the requirements for transaction processing in the Web services environment. In Section 2.3.1, we contrast the ACID properties and the transaction requirements of the Web service environment. In Section 2.3.2, we discuss the problems of the applying the current communication models for distributed transaction to the Web service environment.

2.3.1 Inadequacy of ACID Properties

Unlike traditional transaction processing systems, Web services compositions should not guarantee the ACID properties. Web services compositions require a different and somewhat looser notion of transaction in which individual invocation the Web services may be ACID transactions but the overall composition must have a relaxed set of properties.

The traditional notion of atomicity means that the steps executed during a transaction are completely executed or not at all. There must not be any possibility that only part of a transaction is executed. Such strict notion of atomicity is not reasonable for Web service environment. A composition must exploit the different transaction support provided by Web services. Some Web services can be semantically compensated after they have been successfully executed. If the Web service somehow provides a compensation operation, the composition can exploit it to deal with failures that may occur during its execution. In this case, if the composition specification provides

alternative execution paths to deal with failures, it is possible to avoid the complete abort of the composition when failures occur. For example, consider two Web services ws_1 and ws_2 belonging to the same composition specification *cs*. Moreover, consider that ws_1 is executed after ws_2, according to the composition specification *cs*. Suppose that ws_1 fails during the execution of *cs*. If *cs* provides an alternative execution path for ws_2, then it is possible to undo all services in the path $ws_1 \ldots ws_2$ (backward recovery) that have already been committed by executing compensating services $cws_1 \ldots cws_2$ (assuming that they exist). At this point, it is possible to continue the composition execution through that alternative execution path.

Another aspect of Web services composition that influences atomicity is the presence of many semantically equivalent Web services. During the execution of a composition, whenever a Web service ws_1 fails, if there is another Web service ws_2 supporting an equivalent service with the same semantics of ws_1, then it is possible to continue the execution of the composition even if ws_1 has failed by replacing ws_1 for ws_2. The user-defined notion of alternative execution path, as well as the existence of semantically equivalent Web services, leads to a more flexible notion of atomicity, where the composition may succeed even when some Web services fail, as long as they leave no side effects. This generalization of the *all-or-nothing* property of atomicity is named *2L-guaranteed-termination* property (Section 7.6), which is an extension of the *guaranteed-termination*[3] property present in [5].

Consistency means that a transaction takes the system from one consistent state to another. In order to guarantee consistency across multiple Web services, local transaction subsystems must be monitored to identify applications that violate global dependencies [114]. However, in a Web service environment it is not feasible to guarantee global consistency. Web services are strictly autonomous; it is not possible to interfere with the underlying transaction system of service providers. Moreover, in an environment with a huge number of services, it is not feasible to guarantee global consistency. For example, consistency would require that the state of shared objects be identical across all underlying systems. This would be prohibitively expensive to guarantee in the Internet

[3] While the *2L-guaranteed-termination* property considers the notion of alternative execution path, as well as the existence of semantic equivalent Web services, the *guaranteed-termination* property only considers the notion of alternative execution path; hence, it is not general enough to be used as the correctness criterion for transactional executions of Web services compositions.

with its stateless http protocol. The only level of consistency that must be ensured in the Web service environment is the local Web service consistency. Since each Web service invocation is executed as a single and autonomous transaction, the backend systems of a Web service must ensures local consistency (using any traditional concurrency control protocol). Therefore, a system coordinating the transactional execution of Web services compositions should not provide any level of global consistency beyond what the local backend systems of the constituent Web services provide.

Isolation is concerned with controlling the concurrent execution of transactions to provide the illusion that concurrent transactions are executed in a serial order. Isolation is unnecessarily strict for many applications in Web service environments [39]. This fact is verified by many Internet sites that provide transactions without isolation. For example, sites, such as "Amazon.com", provide transactional semantics in the form of compensation (canceling an order within a given time limit) and do not provide isolation. Moreover, due to Web service autonomy, it might be impossible to maintain the isolation of state updates during the lifetime of a composition since the transaction support of Web services providers may not provide the necessary locking mechanisms. Besides being unnecessarily strict or even impossible in many cases, isolation is also costly because transactions may be long running, and providing isolation for long-running transactions deteriorate the overall system performance. Clearly, providing isolation in the Web service environment, with potentially thousands concurrent users, is not a scalable solution. Moreover, most transaction processing systems work with short time-out intervals for running local transactions. Therefore, Web services coordinated by these transaction-processing systems and executed as part of long-running composition will never succeed due to the local transaction time-out interval.

The fourth property of a transaction is durability. The traditional notion of durability means that when a transaction completes its execution, all of its actions are made persistent, even in presence of system failures. In the same way, durability must be ensured in Web service environments both at composition level and at Web service level. Web service providers must ensure that, after a successful execution of a Web service, its effects are made persistent, even in the presence of system failures. Web services are complete units of computation and they have to handle local failures by their own. A system for coordinating Web services compositions must also provide mechanisms to ensure the durability property at the composition level. A composition execution can

generate intermediate result values that must be persistent, even in the presence of failures. For instance, consider that a Web service for making car reservation has successfully executed in the context of a given composition returning a reservation code as result. Suppose that the system running that composition crashes after receiving the result from the car reservation service. Clearly, for recoverable purposes, it is necessary to guarantee that the value of the reservation code is stored in a durable media.

Analyzing the discussion in the previous paragraphs, the following issues arise:

- Composition executions must ensure only the durability property and a relaxed notion of atomicity, such as the 2L-guaranteed-termination property. Therefore, a transaction model for coordinating Web services Compositions should only ensure a relaxed notion of *failure atomicity* in the executions of Web services compositions. Consequently, the protocols for coordinating the transactional execution of Web services compositions should only provide *recoverability* (termination guarantee). Since isolation and global consistency should not (or cannot) be provided, there is no need of *concurrency control* protocols for coordinating Web services composition executions[4];

- Composition executions rely on Web services to guarantee the durability and guaranteed-termination properties. Therefore, the backend systems of Web services must ensure at least the atomicity and durability properties, locally.

- Web services must expose its transaction support through a standard interface. A composition using such interface can improve its reliability by exploiting the specific transaction support of Web services.

To ensure the failure atomicity of a Web services composition, it is necessary to know the *transaction behavior* of Web services. The transaction behavior defines the level of transaction support that a given Web service exposes. There are two levels of transaction support. The first level consists of Web services that cannot be cancelled after being submitted for execution. Therefore, after the execution of such Web service, it will either commit or abort, and if it commits, its effects cannot be undone. The second level

[4] It is important to note that we are considering a framework for composing strictly autonomous Web services. In an environment where the cooperation between Web services is possible, it would be important to investigate the level of consistency and isolation, which could be achieved. Thus, a transaction model for such environment would have to provide not only recovery protocols but also concurrency control protocols.

consists of Web services that can be aborted or compensated. There are two traditional ways to abort or compensate a previous executed service. One way, named two-phase commit ([46], [72], [71]), is based on the idea that no constituent transaction is allowed to commit unless they are all able to commit. Another way, called compensation, is based on the idea that a constituent transaction is always allowed to commit, but its effect can be cancelled after it has committed by executing a compensating transaction. A transaction model for coordinating Web services compositions must allow the participation of Web services with any level of transactional support, from those that support a 2PC protocol to those that support none transactional protocol.

2.3.2 Inadequacy of Current Distributed Transaction Models

Since some Web services are likely to support a 2PC interface, a general framework for coordinating Web services compositions must be able to interact with distributed transaction coordinators. In Section 2.2.1 and Section 2.2.2, we present a general distributed transaction model used to implement the 2PC protocol. Many distributed transaction systems ([6], [103], [117], [120]) implement some variant of this general distributed transaction model. However, there is a serious drawback with the communication model employed by this kind of distributed transaction model, when considering the Web services environment. The communication model they adopted is the so-called *one-pipe model* [35], where transaction synchronization messages and application messages are sent through the same communication channel, thus using the same communication protocol. For Web services environments, this model is too restrictive. The inherent heterogeneity and autonomy of Web services requires a more flexible model, where application messages are free to use whichever communication protocol they prefer and still have the benefits of transaction support. Similarly, different transaction managers should be able to participate in the two-phase commit protocol using whichever communication protocol they prefer. The *two-pipe model* [35] addresses this communication issue. In this model, the transaction synchronization messages are detached from the application messages. There are two different communication channels, one for transaction synchronization messages and the other for application messages.

This Section raised issues for integrating existent transaction processing systems in a transactional Web service environment. According to these issues, a transaction model for Web services compositions must provide a flexible framework, capable of integrating

different transaction managers without rigid rules on communication protocols and transaction interfaces and without centralized transaction coordination.

2.4 ANALYSIS OF RELATED WORK

As we have shown in Chapter 1, many different technologies are involved in transactional composition of Web services and thus they have influenced the development of WebTransact. In the next sections, we discuss each one of these technologies. Section 2.4.1 reviews the works on mediator systems, whose ideas serve as the basis for the proposed solution on aggregating semantically equivalent Web services. Section 2.4.2 provides an analysis of the existent works on transaction support for distributed computing. Section 2.4.3 reviews the work on Extended Transaction Models. Next, Section 2.4.4 presents recent related works on service composition. Finally, Section 2.5 reviews the current Web services technology.

2.4.1 Mediator Technology

A generic architecture for the integration of information sources involves the systems based on Mediators (*Intelligent Information Integration (I3) Mediation*) [148]. Such architecture is based on components named Mediators. According to Wiederhold [148], a Mediator is a software component, which explores the knowledge represented in a set or subset of data, to generate information for applications residing in an upper layer. Each Mediator encapsulates the representation of multiple data sources and provides the functionality of uniform access to data. Thus, this component resolves the conflicts that commonly arise in such environments, like those concerning knowledge representation (different schemas).

The user accesses system data through queries, written in a global language, submitted to the Mediator. Then, the Mediator transforms the queries into sub-queries and sends them to the local data repositories.

The sub-queries generated by Mediators must be translated from the global language into the query language of each data repository. Wrappers are responsible for this functionality. These components map the sub-queries written in the global language into the local query language and return the reformatted responses to the appropriate

Mediator. This component solves problems related to differences in the query expressiveness of each repository.

Several projects based on this model have been developed by the international research community, such as the projects TSIMMIS [42], Garlic [19], DISCO [132], HIMPAR ([104], [105], [106]) and DIOM [77]. These projects aim at integrating structured and non-structured (with no data schema) data sources. They also deal with issues related to the mismatch in querying power of the different data sources and propose several techniques to enable the reformulation of queries to resolve this mismatch.

Wrapper mediator systems ([19], [42], [50], [51], [75], [104], [128], [132], [134], [135], [150]) have been successfully developed to mediate the query capability of remote (database or non-database) sources and to provide information integration at the mediator level. Typically, these projects support transactions within the mediator but the transaction semantics is not extended to the remote services. Supporting a transaction service that is extended to the remote servers is critical if we wish to implement E-business services based on wrapper mediator systems. Therefore, current technology on mediators is prepared to deal with data and query integration, but not services in general.

2.4.2 Transaction Support for Distributed Computing

There are related works on distributed transactions in three areas, heterogeneous DBMS, workflow management systems (WMS), and composite systems.

Research in heterogeneous DBMS is related to architectures of heterogeneous distributed DBMS, and they investigate transaction support in the face of incompatible concurrency control mechanisms and/or uncooperative transaction managers in the underlying DBMS. A survey of research issues is in ([32], [108]). Transaction support in heterogeneous DBMS operates at two levels. One is the local level of some pre-existing transaction manager of the component DBMS - the local transaction manager (LTM) and the second is at the global level - a global transaction manager (GTM). The existence of both the GTM and the LTMs introduces distributed control for implementing transactions. The tasks that must be managed by this distributed control mechanism include managing the autonomy of individual remote servers, and the consistency requirements of global transactions. Much of the research in heterogeneous DBMS addresses the issue of data consistency in global transactions, while preserving different aspects of remote server

autonomy. This capacity is very useful in Web services compositions. However, the mechanisms for supporting global consistency relies on specific properties from the underlying DBMS, which are not always present (or exposed) on the underlying systems that support Web services.

Research on workflow management systems (WMS) ([56], [73]) has been categorized as multi-database and extended transactions ([7], [33], [41], [145]); active DBMS and rule-based approaches ([29], [30],[94]); combinations of the above [44]; and office and/or process automation ([79], [82], [90]). Workflow supports the specification of intra-transaction and inter-transaction state dependencies. It provides solutions that focus on correctness based on serializability, visibility, cooperation, and temporal dependencies. Similarly, to heterogeneous DBMS, the majority of workflow management systems exploit the support provided by the underlying systems for coordinating the transaction execution of workflows. Therefore, the solutions for coordinating transactions in the workflow area are not flexible enough to be directly applied in the Web services environment.

Research on composite systems ([1], [3], [5], [100], [113], [114]) is related to transaction process over a hierarchical set of remote systems. Composite systems have an architecture with two layers. The top layer controls the execution of transactional processes, as specified in process programs. Each one of these process programs is a set of partially ordered activities. Each activity, in turn, corresponds to a conventional transaction executed in a transactional application. Hence, activities are, by definition, atomic and therefore terminate either committing or aborting. The bottom layer of the system model is formed by the universe of all available independent transactional applications (subsystems). Each of these subsystems has to provide serializable executions and avoid cascading aborts (ACA) [14]. The concurrent execution of transactional processes is controlled by a transactional process manager (PM), which is responsible for scheduling the invocation of transactions in the underlying applications. Composite systems allow subsystems with dissimilar transaction capabilities to participate in a global transaction. When a subsystem does not provide transaction support (i.e., it supports non-atomic activities), it is required that such subsystem provides interfaces to allow a coordination agent to undo all effects of a failed activity. Therefore, the transaction coordination of composite systems requires cooperation of its subsystems. Since the Web services are strict autonomous, the solutions for coordinating transactions

in the composite systems area cannot be directly applied in the Web services environment.

While heterogeneous DBMS, WMS, and composite systems, support distributed transactions, they do not consider those aspects of transaction support needed to mediate among the service capabilities of autonomous Web service providers, i.e., the transaction coordination of these systems requires different levels of cooperation of the underlying systems, which cannot be assumed in Web services environment. WebTransact uses knowledge of the transaction capabilities of Web service providers, and semantic knowledge of composition execution flow, to provide a safe execution plan corresponding to a distributed transaction. Moreover, all these works were developed without considering the XML-based standards for the Web service technology.

2.4.3 Extended Transaction Models

An important step towards the evolution of transaction models was the extension of the flat (single level) transaction structure to multi-level structures. A Nested Transaction [93] is a set of subtransactions that may recursively contain other subtransactions, thus forming a transaction tree. A child transaction can start after its parent has started and a parent transaction can only terminate after all its children terminate. If a parent transaction is aborted, all its children are aborted. Nested transactions provide full isolation on the global level, but they permit increased modularity, finer granularity of failure handling, and a higher degree of intra-transaction concurrency than the traditional transactions. Open Nested Transactions [146] relax the isolation requirements by making the results of committed subtransactions visible to other concurrently executing nested transactions. They also permit the modeling of higher-level operations and the exploiting of their application-based semantics, especially the commutativity of operations.

In addition to the extension of internal transaction structure, Extended Transaction Models focus on selective relaxation of atomicity or isolation and may not require serializability as a global correctness criterion. They frequently use inter-transaction execution dependencies that constrain scheduling and execution of component transactions. Many of these models were motivated by specific application environments and they attempt to exploit application semantics.

Most of the Extended Transaction Models use some form of compensation. A subtransaction can commit and release the resources before the (global) transaction successfully completes and commits. If the global transaction later aborts, its failure atomicity may require that the effects of already committed subtransactions be undone by executing compensating subtransactions. Relaxing the isolation of distributed transactions may cause violation of global consistency (global serializability), since other transactions may observe the effects of subtransactions that will be compensated later ([43], [68]). The concept of a horizon of compensation in the context of multi-level activities has been proposed in [69]. Under this model, a child operation can be compensated only before its parent operation commits. Once the parent operation commits, the only way to undo the effects of a child operation is to compensate the entire parent operation.

The concept of *Sagas* was introduced in [43] (further extended in [40]) to deal with long-running transactions. A saga consists of a set of ACID subtransactions $T_1,...,T_n$ with a predefined order of execution, and a set of compensating subtransactions $CT_1,...,CT_n$. corresponding to $T_1,...,T_{n-1}$. A saga completes successfully if the subtransactions $T_1,..., T_n$ have committed. If one of the subtransactions, for instance T_j, fails, then committed subtransactions $T_1,...,T_{j-1}$ are undone by executing compensating subtransactions $CT_1,...,CT_{j-1}$. Sagas relax the full isolation requirements and increase inter-transaction concurrency.

Flexible Transactions ([33], [7]) have been proposed as a transaction model suitable for a multidatabase environment. A flexible transaction consists of a set of tasks, with a set of functionally equivalent subtransactions for each task and a set of execution dependencies on the subtransactions, including failure dependencies, success dependencies, or external dependencies. To relax the isolation requirements, flexible transactions use compensation and relax global atomicity requirements by allowing the transaction designer to specify acceptable states for termination of the flexible transaction, in which some subtransactions may be aborted. IPL [26] is a language proposed for the specification of flexible transactions with user-defined atomicity and isolation. It includes features of traditional programming languages such as type specification to support specific data formats that are accepted or produced by subtransactions executing on different software systems, and preference descriptors with logical and algebraic formulae used for controlling commitments of transactions.

Other works on Extended Transaction Models can be found in [31], [64], and [110]. The transaction model of WebTransact is inspired in the ideas of compensation and user-defined atomicity from Extended Transaction Models. However, the transaction model of WebTransact provides more flexibility since it considers the dissimilar transaction behavior of Web services. Moreover, in order to support the requirements of Web services compositions, the transaction model of WebTransact supports a very relaxed notion of the ACID transaction properties. Only the durability is fully supported while a weak notion of atomicity is supported. These features make the transaction model of WebTransact unique among the existent transaction models for distributed computing.

2.4.4 Web services composition

The area of service composition is attracting strong interest from both academia and industry.

The Web Services Flow Language (WSFL) [76] is an XML language for the description of Web services compositions as part of a business process definition. It was designed by IBM to be part of the Web service technology framework and it relies and complements existing specifications like SOAP, WSDL, and UDDI. WSFL considers two types of Web services compositions. The first type, named *flow model*, specifies the appropriate usage pattern of a collection of Web services, in such a way that the resulting composition describes how to achieve a particular business goal; typically, the result is a description of a business process. The second type, named *global model*, specifies the interaction pattern of a collection of Web services; in this case, the result is a description of the overall partner interactions.

In a flow model, a composition is created by describing how to use the functionality provided by the collection of composed Web services. This is also known as flow composition, orchestration, or choreography of Web services. WSFL models these compositions as specifications of the execution sequence of the functionality provided by the composed Web services. Execution orders are specified by defining the flow of control and data between Web services. Flow models can be used particularly to model business processes or workflows based on Web services.

In a Global model, no specification of an execution sequence is provided. Instead, the composition provides a description of how the composed Web services interact with

each other. The interactions are modeled as links between endpoints of the Web services interfaces, each link corresponding to the interaction of one Web service with an operation of another Web service interface.

The Business Process Modeling Language (BPML) [8] is a meta-language for the modeling of business processes, just as XML is a meta-language for the modeling of business data. BPML provides an abstract execution model for collaborative and transactional business processes based on the concept of a transactional finite-state machine. BPML business process definition is an ordered collection of activities: simple, complex, and process activities. Simple activities are made of messages (incoming and outgoing), they constitute the message and data flow of a business process. Complex and process activities represent the control flow of the business process definition.

XLANG [129] is an extension of WSDL that provides both the model of an orchestration of services as well as collaboration contracts between orchestrations. In XLANG, a process definition is specified within a service definition. Such process definition specifies the behavior of the service. A service with a behavior represents an interaction spanning many operations; the incoming and outgoing operations of the XLANG service represent interactions with other services, therefore sequencing the operations of a given service is equivalent to orchestrating a series of services. The interaction has a well-defined beginning and end.

BPML, and XLANG support transactional semantics at the Web service composition level. BPML supports two transaction models: coordinated and extended. They both provide an "all-or-nothing" guarantee for complex interactions between multiple participants. In a coordinated transaction, all participants agree to either complete the transaction or abort it. This model relies on a two-phase commit protocol and it is also known as a closed, flat transaction due to its ability to support isolation. The extended transaction model relaxes the isolation requirement. There are two possible modes of recovery for such a transaction: backward recovery, where the transaction initiates some compensating activities that will cancel the effects of the failed transaction, and forward recovery, where the business process instance is allowed to continue its execution taking into account that the transaction failed. Compensating activities are used wherever the execution of an activity cannot be rolled back (such as when an order is shipped). These transactions are also known as open nested transactions or Sagas. This model supports

transaction interleaving with arbitrary nesting, in other words, transaction may be composed of other transactions. XLANG has introduced the notion of a context for local declaration of correlation sets and port references, exception handling, and transactional behavior. A context provides and limits the scope over which declarations, exceptions, and transactions apply. XLANG supports open transactions, but unlike BPML, it does not support coordinated transactions. XLANG transactions follow the model of long-running transactions, which are associated with compensating actions in case the transaction fails.

More recently, the HP Labs propose the Web Services Conversation Language (WSCL) [136]. WSCL allows defining abstract interfaces of Web services, i.e., the business level conversations or public processes supported by a Web service. WSCL specifies the XML documents being exchanged, and the allowed sequencing of these document exchanges. WSCL may be used in conjunction with other service description languages like WSDL.

Although all these works allow the composition of Web services, none of them explicit deals with mediating the dissimilar transaction behavior of Web service nor consider the problem of aggregating semantically equivalent Web services. The transaction support offered by XLANG and BPML is restricted to the *specification* of the desired transaction properties of a given composition; they do not deal with the problem of coordinating the execution of the resulting transactions. Still, these languages do not address the problem of representing the dissimilar transaction behavior of Web services. They assume that the underlying system provide the necessary transaction support.

2.5 CURRENT WEB SERVICE TECHNOLOGY

During the elaboration of this work, a *de facto* standard for describing services in the web arose. This standard, named Web Services Description Language (WSDL) [137], specifies service interfaces, the protocol to communicate with them, and their Internet location. The WebTransact Model uses WSDL for describing remote services and proposes an extension to it to allow the description of the transaction behavior of remote services. Next, we provide an explanation of WSDL. The extension to WSDL proposed by this work is described from Chapter 4 to Chapter 6.

Web Services Description Language (WSDL) is an XML language [138] for describing Web services in a standardized way. This standard description enables the

automated generation of proxies for Web services independent of language and platform. In the same way middleware systems like COM [84] and CORBA [97] uses interfaces, a WSDL document is a contract between service providers (servers) and their clients. A WSDL document defines services as collections of network endpoints, or ports [137]. In WSDL, there is a separation between the *abstract* definition of messages and their *concrete* network implementation. This allows the reuse of abstract definitions of messages and port types. Messages are abstract descriptions of the data being exchanged, and port types are abstract collections of operations. The concrete protocol and data format specification for a particular port type defines a reusable binding. A port is specified by associating a network address with a reusable binding. A service is defined as a collection of ports. The following six major (XML) elements define a Web service:

1) Abstract Definitions:

- *Type* is a machine and language independent type definition, which provides data type definitions to describe the messages exchanged.

- *Message* describes data being exchanged through a typed definition. A message consists of logical parts, each of which is associated with a definition within some type system.

- *PortType* is a set of operations supported by one or more endpoints. Each operation refers to input and output messages.

2) Concrete Definitions:

- *Binding* defines a protocol and data format specifications for the operations and messages defined by a particular portType.

- *Port* is a single endpoint defined as a combination of a binding and a network address.

- *Service* is a collection of related ports.

Figure 2.6 shows the relationship among the elements of a WSDL document. Messages make use of type definitions from the Types section; PortTypes make use of message definitions from the Messages section; Bindings refer to port types defined in the PortTypes section; and Services refer to bindings defined in the Bindings section.

PortTypes and Bindings contain *Operation* elements, and Services contain Port elements. An Operation element is a description of an action supported by the Service. Operation elements in the PortTypes section are modified or further described by Operation elements in the Bindings section.

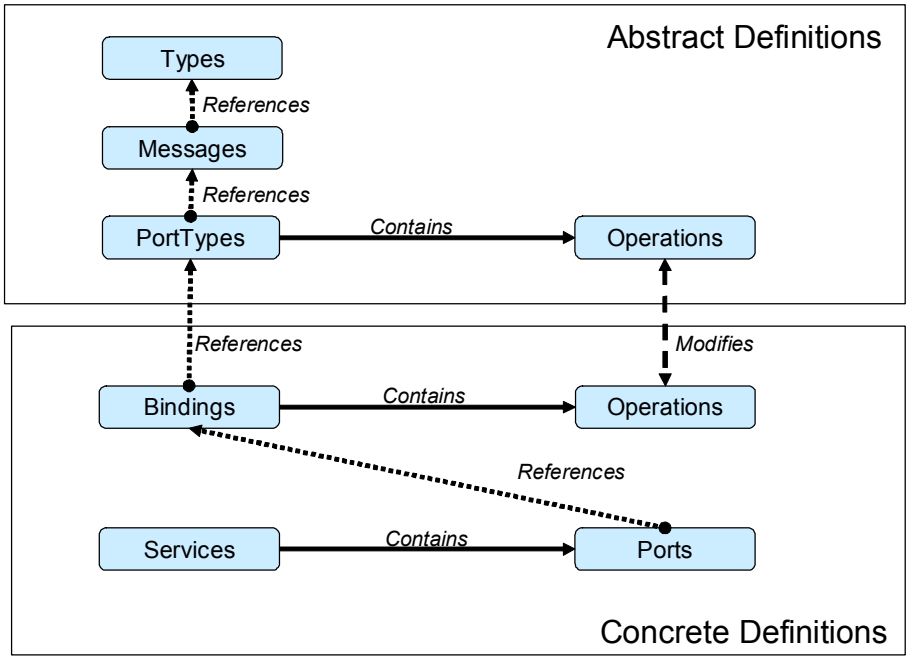


Figure 2.6 - WSDL document elements.

Note that there can be only one Types section or no section at all. All other sections can have zero, one, or multiple parent elements. For example, the Messages section can have zero or more message elements. The WSDL Schema requires that all sections appear in a specific order: Types, Message, PortType, Binding, and Service. Each abstract section may be in a separate file by itself and imported into the main document. This is done through the *Import* element.

These elements are described in detail in Section 2.6. It is important to note that WSDL does not introduce a new type definition language. WSDL recognizes the need for rich type systems for describing message formats, and supports the XML Schemas specification (XSD) ([138], [141], [142]) as its canonical type system. However, since it is unlikely to expect a single type system grammar to be used to describe all message formats ever, WSDL allows using other type definition languages through extensibility. In addition, WSDL defines a common binding mechanism. This is used to attach a

specific protocol or data format or structure to an abstract message, operation, or endpoint. It allows the reuse of abstract definitions.

In addition to the core service definition framework, the WSDL specification [137] introduces specific *binding extensions* for the SOAP 1.2 [144], HTTP [37], and MIME [38] protocols. These language extensions are layered on top of the core service definition framework.

2.6 WSDL DOCUMENT STRUCTURE

A WSDL document is simply a set of XML definitions. There is a *Definitions* element at the root, and definitions inside [137]:

```
<-- The characters appended to elements and attributes indicates    -->
<-- the number of allowed occurrences of each element or            -->
<-- attribute as follows:                                           -->
<-- “?” (0 or 1); “*” (0 or more); “+” (1 or more)                  -->

<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>

  <import namespace="uri" location="uri"/>*

  <wsdl:documentation…/>?

  <--    Types Element Definitions    -->*
  <--    Messages Element Definitions        -->*
  <--    PorTypes Element Definitions        -->*
  <--    Bindings Element Definitions        -->*
  <--    Services Element Definitions        -->*
  <--    Extensibility Element Definitions   -->*

</wsdl:definitions>
```

WSDL documents can be assigned an optional `name` attribute that serves as a lightweight form of documentation. Optionally, a `targetNamespace` attribute of type `URI` may be specified. WSDL allows associating a namespace with a document location using an `import` statement. The use of the `import` element allows the separation of the different elements of a service definition into independent documents, which can then be imported as needed. This technique helps writing clearer service definitions, by separating the definitions according to their level of abstraction. It also maximizes the ability to reuse service definitions. WSDL uses the optional `wsdl:documentation` element as a container for human readable documentation. The content of this element is arbitrary text and elements. The documentation element can appear inside any WSDL language element.

The root element `definitions` has five child elements that actually describe the service. These elements are explained in the next subsections.

2.6.1 Types Section

The types element comprises the Types section. The types element encloses data type definitions relevant for defining the exchange of messages. This section may be omitted if there are no data types that need to be declared. The grammar for types is as follows:

```
<wsdl:types>?
  <wsdl:documentation />?
  <xsd:schema…/>*
    <-- definition of new elements --> *
  </wsdl:types>
```

The role of this element can be compared to that of the Schema element of the XML Schema language [141].

2.6.2 Message Section

The `message` elements comprise the Messages section. If we consider operations as functions, then a `message` element defines the parameters to that function. The grammar for a message is as follows:

```
<wsdl:message name="nmtoken> *
  <wsdl:documentation…/>?
  <part name="nmtoken" element="qname"? type="qname"?/> *
</wsdl:message>
```

Each `part` child element in the `message` element corresponds to a parameter. Input parameters are defined in a single `message` element, separate from output parameters, which are in their own `message` element. Parameters that are both input and output have their corresponding `part` elements in both input and output `message` elements. Each `part` element has a name and type attribute, just as a function parameter has both a name and a type. The type of a `part` element can be an XSD element (`element`) or an XSD `simpleType` or `complexType`. When used for document exchange, WSDL allows the use of `message` elements to describe such documents.

Message definitions are always considered an abstract definition of the message content. A message binding describes how the abstract content is mapped to a concrete format. However, in some cases, the abstract definition may match the concrete representation very closely or exactly for one or more bindings, so those bindings will supply little or no mapping information. However, another binding of the same message definition may require extensive mapping information. For this reason, it is not until the binding is inspected that one can determine "how abstract" the message really is.

2.6.3 Port Types Section

The `portType` elements comprise the Port Types section. A PortType is a named set of abstract operations and the abstract messages involved. The grammar for a port type is as follows:

```
<wsdl:portType name="nmtoken">*
  <wsdl:documentation…/>?
  <wsdl:operation name="nmtoken" parameterOrder="nmtoken"?>*
    <wsdl:documentation…/> ?
    <wsdl:input name="nmtoken"? message="qname">?
      <wsdl:documentation…/> ?
    </wsdl:input>
    <wsdl:output name="nmtoken"? message="qname">?
      <wsdl:documentation…/> ?
    </wsdl:output>
    <wsdl:fault name="nmtoken" message="qname"> *
      <wsdl:documentation…/> ?
    </wsdl:fault>
  </wsdl:operation>
</wsdl:portType>
```

The `operation` elements within a `portType` element define the syntax for calling all methods in the `portType`. Each `operation` element declares the name of the operation and its parameters using former declared `message` elements. Each `portType` element groups together a number of related operations in much the same way as a CORBA [97] or COM [84] interface groups a number of related methods. There can be zero, one, or more `portType` elements in the Port Types section. Because abstract port type definitions can be placed in a separate file, it is possible to have zero `portType` elements in a WSDL file. The optional `fault` elements specify the abstract message format for any error messages that may be output as the result of the operation (beyond those specific to the protocol). The `portType` elements allow the specification of four kinds of transmission primitives (*operations*) that an endpoint can support:

1) **One-way**. The endpoint receives a message.

2) **Request-response**. The endpoint receives a message, and sends a correlated message.

3) **Solicit-response**. The endpoint sends a message, and receives a correlated message.

4) **Notification**. The endpoint sends a message.

Although request/response or solicit/response are logically correlated in the WSDL document, a given binding describes the concrete correlation information. For example,

the request and response messages may be exchanged as part of one or two actual network communications.

Operations do not specify whether they are to be used with RPC-like bindings or not. However, when using an operation with an RPC-binding, it is useful to be able to capture the original RPC function signature. For this reason, a request-response or solicit-response operation may specify a list of parameter names through the `parameterOrder` attribute (of type `nmtokens`). The value of the attribute is a list of message part names separated by a single space.

2.6.4 Bindings Section

A binding defines message format and protocol details for operations and messages defined by a particular `portType`. There may be any number of bindings for a given `portType`. The grammar for a binding is as follows:

```
<wsdl:binding name="nmtoken" type="qname">*
    <wsdl:documentation…/>?
    <-- extensibility element (1) --> *
  <wsdl:operation name="nmtoken">*
    <wsdl:documentation…/> ?
    <-- extensibility element (2) --> *
    <wsdl:input name="nmtoken"?> ?
      <wsdl:documentation…/> ?
      <-- extensibility element (3) -->
    </wsdl:input>
    <wsdl:output name="nmtoken"?> ?
      <wsdl:documentation…/> ?
      <-- extensibility element (4) --> *
    </wsdl:output>
    <wsdl:fault name="nmtoken"> *
      <wsdl:documentation…/> ?
      <-- extensibility element (5) --> *
    </wsdl:fault>
  </wsdl:operation>
</wsdl:binding>
```

A `binding` references the `portType` that it binds using the `type` attribute. The binding's `operation`, `input`, `output`, and `fault` elements are correlated with the corresponding `portType` elements using the `name` attribute of each element, which behave exactly as within `portType`. Binding extensibility elements are used to specify the concrete grammar for the `input` (3), `output` (4), and `fault` messages (5). Binding information within an operation (2) as well as within the binding itself (1) may be specified. A binding specifies exactly one protocol.

The Binding section is where the protocol, serialization, and encoding are fully specified. Whereas the Types, Messages, and Port Types sections specify abstract data

content, the Binding section specifies the physical details of data transmission. The Binding section concretizes the abstractions made in the first three sections.

The separation of binding specifications from data and message declarations means that service providers who engage in the same type of business can agree on a set of operations. Each provider can then differentiate from one another by providing custom bindings. In addition, WSDL has an *import* construct so that the abstract definitions can be put in their own file, separate from the Bindings and Services sections, which can be distributed among service providers for whom the abstract definitions will have been established as a standard. For example, banks can define a standard set of banking operations accurately described in an abstract WSDL document. Then, each bank is still free to "customize" an underlying protocol, serialization optimizations, and encoding.

2.6.5 Service Section

A service groups a set of related ports as follows:

```
<wsdl:service name="nmtoken"> *
  <wsdl:documentation…/>?
  <wsdl:port name="nmtoken" binding="qname"> *
    <wsdl:documentation…/> ?
    <-- extensibility element -->
  </wsdl:port>
    <-- extensibility element -->
</wsdl:service>
```

Each `port` element associates a location with a `binding` in a one-to-one fashion. If there is more than one `port` element associated with the same `binding`, then the additional URL locations can be used as alternatives. There can be more than one `service` element in a WSDL document. There are many situations when multiple `service` elements are useful. One of them is to group together ports according to URL destination. Thus, it is possible to redirect a given stock quote request simply by using another `service`. Another use of multiple `service` elements is to classify the ports according to the underlying protocol. Within one WSDL document, the service `name` attribute distinguishes one service from another. Because there can be several ports in a service, a `port` element also has a `name` attribute.

Both the Bindings and Services sections comprise the description of concrete elements of a WSDL document.

2.6.6 WSDL Document Example

The following example shows the WSDL definition of a simple service providing car rental rates (Figure 2.7). The service supports a single operation called `GetRate`, which is deployed using the SOAP 1.1 protocol over HTTP. This operation receives a car class of type `carClass`, and returns the price as a float.

```
<?xml version="1.0" encoding="UTF-8" ?>
<definitions name="CarRental"
  targetNamespace="http://example.com.br/carRental.wsdl"
  xmlns:wsdlns="http://example.com.br/carRental.wsdl"
  xmlns:typens="http://example.com.br/carRental.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd=="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

<wsdl:documentation Begin of Types Section />
<types>
  <schema targetNamespace="http://example.com.br/carRental.xsd"
          xmlns="http://www.w3.org/2001/XMLSchema"
  element name="carClass">
    <simpleType>
      <restriction base=”string”>
        <enumeration value=”economy” />
        <enumeration value=”sport” />
        <enumeration value=”luxury” />
        <enumeration value=”family” />
      </restriction>
    </simpleType>
  </element>
  </schema>
</types>

<wsdl:documentation Begin of Messages Section />
<message name="GetRate.input">
  <part name="pCarClass" type=" typens:carClass"/>
</message>
<message name="GetRate.output">
  <part name="result" type="xsd:float"/>
</message>

<wsdl:documentation Begin of PortTypes Section />
<portType name="GetRatePortType">
  <operation name="GetRate" parameterOrder="pCarClass" >
    <input message="wsdlns:GetRate.input"/>
    <output message="wsdlns:GetRate.output"/>
  </operation>
</portType>

<wsdl:documentation Begin of Bindings Section />
<binding name="GetRateBinding" type="wsdlns:GetRatePortType">
<soap:binding style="rpc"
     transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetRate">
    <soap:operation soapAction="http://example.com.br/CarRental.GetRate"/>
      <input>
        <soap:body use="encoded" namespace="http://example.com.br/CarRental/"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </input>
      <output>
        <soap:body use="encoded" namespace="http://example.com.br/CarRental/"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </output>
  </operation>
</binding>

<wsdl:documentation Begin of Services Section />
<service name="CarRentalService">
  <port name="CarRentalPortBr" binding="wsdlns:GetRateBinding">
    <soap:address location="http://example.com.br/CarRental/GetRate.asp"/>
  <port name="CarRentalPortFr" binding="wsdlns:GetRateBinding">
    <soap:address location="http://example.com.fr/CarRental/GetRate.asp"/>
```

```
  </port>
</service>
</definitions>
```

Figure 2.7 - A WSDL document of a request/response operation using SOAP binding.

In the sample WSDL file of Figure 2.7, the `targetNamespace` for `definitions` is `http://example.com.br/carRental.wsdl`. This means that all names declared in this WSDL document belong to this namespace. The `schema` element has its own `targetNamespace` attribute with a value of `http://example.com.br/carRental.xsd` so that all names defined in this `schema` element belong to this namespace instead of the main target namespace.

There is one application-specific type declared, named `carClass`. The `carClass` element in the Types section has embedded in it an anonymous enumeration type whose possible values are `economy`, `sport`, `luxury`, and `family`. This element is used in the `GetRate.input` message to define the type of the `pCarClass` part of this message. The `message` element named `GetRate.input` defines the request part of the `GetRate` operation (input parameters); while the message element `GetRate.output` defines the response part of that operation (output parameters). The example of Figure 2.7 shows only one `portType` element - `GetRatePortType` - with one `operation` element, named `GetRate`. The `GetRate` element has two child elements: the `input`, and `output` elements. The `message` attributes in each `input` and `output` element refers to the relevant `message` element in the Messages section. Thus, the whole `portType` element in the sample is equivalent to the following C function declaration:

```
float GetRate(CarClass pCarClass);
```

There is one binding element named `GetRateBinding`, which has a `type` attribute that refers to the `portType` element named `GetRatePortType`. The `soap:binding` element specifies the style and transport used. The `transport` attribute refers the `http://schemas.xmlsoap.org/soap/http` namespace, which indicates that the HTTP SOAP protocol is used. The `style` attribute value - `rpc` - indicates that the default style of the operations is RPC-oriented (messages containing parameters and return values). There is one `operation` element with the name `GetRate`. The `soap:operation` element within the `operation` element has a `soapAction` attribute, which is a URI. The `soapAction` attribute - `http://example.com.br/`

`CarRental.GetRate` - is a SOAP-specific URI that is included in the SOAP message. The resulting SOAP message has a `SOAPAction` header and this URI in the `soap:operation` element becomes its value. The `soapAction` attribute is required for HTTP binding but should not be present for non-HTTP binding. The `soap:operation` element can also contain another attribute, the `style` attribute used when it is necessary to override the style specified in the `soap:binding` element for this particular operation.

The `operation` element `GetRate` contains `input`, and `output` elements, which all correspond to the same elements in the Port Types section. In the example of Figure 2.7, inside the `input` element is a `soap:body` element that specifies what is inserted into the `body` of the resulting SOAP message. This element has the following attributes:

1) `Use`. This is for specifying whether the data is `encoded` or `literal`. `Literal` means that the resulting SOAP message contains data formatted exactly as specified in the abstract definitions (Types, Messages, and Port Types sections). `Encoded` means that the `encodingStyle` attribute determines the encoding.

2) `Namespace`. Each SOAP message body can have its own namespace to prevent name conflicts. The URI specified in this attribute is used verbatim in the resulting SOAP message.

3) `EncodingStyle`. For SOAP encoding, this attribute must have the URI value of `http://schemas.xmlsoap.org/soap/encoding`.

Finally, the `service` element `CarRentalService` groups together two ports: `CarRentalPortBr` and `CarRentalPortFr`. Both ports reference the same `binding` element `GetRateBinding`. Therefore, both ports provide semantically equivalent behavior and the same transport and format of messages. The difference between the two ports is the location of the service provider. The `port` element `CarRentalPortBr` has the `location` attribute of its `soap:address` element as `http://example.com.br/CarRental/GetRate.asp`; this `location` attribute references a site in Brazil while the `port` element `CarRentalPortFr` has the `location` attribute of its `soap:address` element as `http://example.com.fr/CarRental/GetRate.asp`; this `location` attribute references a site in France. This difference allows a consumer of the WSDL document to

choose one of these ports to communicate, based on the distance criteria. Note that we are considering both sites having the same content. If such assumption is not true, then those sites are not *content* equivalent, and thus one cannot be transparently used in the place of the other. Therefore, even though WSDL provides location transparency when using a service, it does not solve problems regarding dissimilar message structures between semantically equivalent Web services or problems related to the dissimilar content of Web services. These issues are both addressed by WebTransact (Section 5.2).

2.6.7 Bindings Extension with WSDL

WSDL is more than just a language to specify interface contracts between client applications and service providers. Beyond the interface contract, WSDL also specifies the high-level communication protocol that is used between a particular client and a particular server. A WSDL document specifies not only the operations supported by a specific server but also how to implement the serialization of the operation parameters. In other words, given any kind of operation, through a WSDL document specification it is possible to define the number of parameters, the type of each parameter, and, by the *binding* mechanism, how those parameters are to be sent through the wire (serialization).

The WSDL 1.1 [137][136] introduces specific binding extensions for the following protocols and messages formats: SOAP 1.2 [144], HTTP GET/POST [37], and MIME [38]. These binding extensions are layered on top of the core service definition framework of WSDL. Therefore, it is possible to define other binding extensions for any kind of protocol or message format.

2.6.8 Wire Format for the WSDL Example

The XML messages shown in Figure 2.8 and Figure 2.9 are sent and received as a result of parsing the sample WSDL document shown in Figure 2.7.

```
POST /CarRental HTTP/1.1
Host: www.carRentalServer.com.br
Content-Type: text/xml; charset=”utf-8”
Content-lengh: nnn
SOAPaction: “http://example.com.br/CarRental.GetRate”
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <m:GetRate xmlns:m="http://example.com.br/CarRental/">
      <pCarClass>Economy</pCarClass>
    </m:GetRate>
  </SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>
```

Figure 2.8 - SOAP message embedded in a HTTP request that is sent from the client to make a function call `GetRate("Economy")`.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-lengh: nnn
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <SOAPSDK1:GetRate.output xmlns:SOAPSDK1="http://example.com.br/CarRental/">
      <result>60.50</result>
    </SOAPSDK1: GetRate.outPut>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 2.9 - SOAP message embedded in HTTP response from server.

Both the function call message and its response are valid XML. A SOAP message consists of an `Envelope` element that contains an optional `Header` element and one `body` element, at a least. Both sent and received messages have a single `Body` element in the main `Envelope` element. The message body of the RPC function call has an element named after the operation name `GetRate`, while the response body has a `GetRate.output` element. The `GetRate` element has one part, `pCarClass`, which is the single argument, as described in the sample WSDL. The `GetRate.output` has a single `result` part. Note how the encoding style, envelope, and message namespace is as prescribed in the Bindings section of the WSDL document in Figure 2.7.

2.6.9 WSDL Lacking Features for developing Transactional Web services compositions

The WSDL framework provides the necessary communication model for message exchanges in the Web services environment. However, communication is only part of the problem when considering the building of reliable Web services compositions. The first lacking feature of WSDL is the definition of elements for describing the dissimilar transaction behavior of Web services. The second is the definition of mechanisms for homogenizing heterogeneous, but semantically equivalent, Web services. The third is the definition of elements for describing the dissimilar content capability of semantically equivalent Web services. The fourth, which is clearly out of the scope of WSDL, is the definition of mechanisms for building value-added service using existent Web services, i.e., the composition of Web services. Therefore, in order to develop a framework for

building reliable Web services compositions we have extended WSDL with new elements. Such extension is implemented by a new XML-based language, named *Web Services Transaction Language* (WSTL), which is described in the next chapters.

3. Overview of WebTransact

This chapter presents an overview of the WebTransact framework. First, we present a general picture of the WebTransact architecture. Next, we briefly explain the components of that architecture. Finally, we sketch the operation semantics of WebTransact. The details of the components of the WebTransact framework are presented in Chapter 5 and Chapter 6, while the operational semantics of WebTransact is discussed in its details in Chapter 7.

3.1 ARCHITECTURE

As shown in Figure 3.1, WebTransact enables Web services composition by adopting a multilayered architecture of several specialized components. Application programs interact with *composite mediator services* written by composition developers. Such compositions are defined through transaction interaction patterns of *mediator services*. Mediator services provide a homogenized interface of (several) semantically equivalent *remote services*. Remote services integrate Web services providing the necessary mapping information to convert messages from the particular format of the Web service to the mediator format.

The WebTransact architecture encapsulates the message format, content, and transaction behavior of multiple Web services and provides different levels of value-added services. First, the WebTransact architecture provides the functionality of uniform access to multiple Web services. Remote services resolve conflicts involving the dissimilar representation of knowledge from different Web services, and conflicts due to the mismatch in the content capability of each Web service. Besides resolving structural and content conflicts, remote services also provide information on the interface and the transaction behavior supported by Web services. Second, Mediator services integrate semantically equivalent remote services providing a homogenized view on heterogeneous Web services. Finally, transaction interaction patterns are built on top of those mediator services generating composite mediator services that can be used by application programs or exposed as new complex Web services

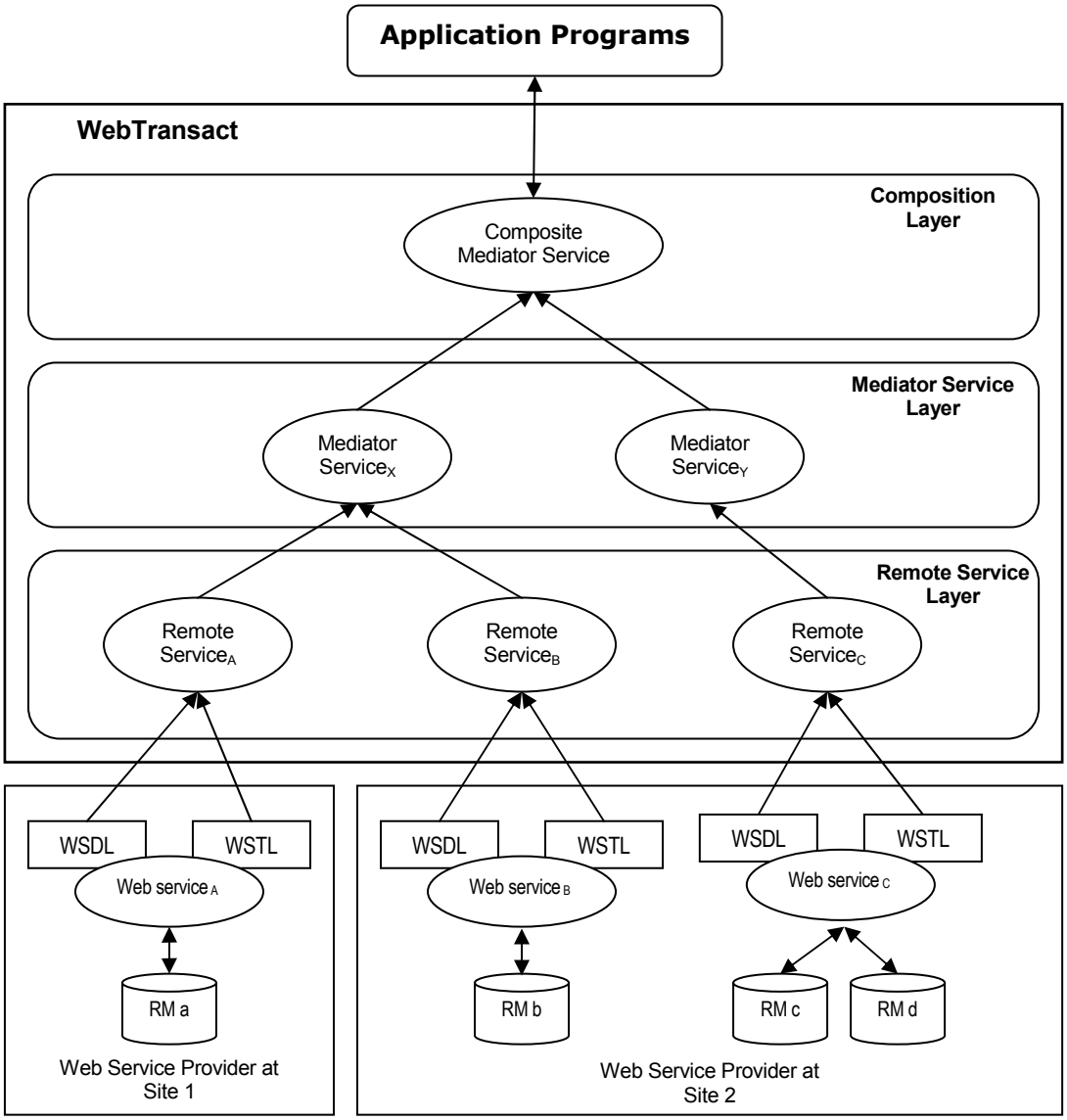


Figure 3.1 - WebTransact Architecture.

WebTransact integrates Web services through two XML-based languages: Web Service Description Language (WSDL), which is the current standard for describing Web service interfaces, and Web Service Transaction Language (WSTL), which is our proposal for enabling the transactional composition of heterogeneous Web services. WSTL is built on top of WSDL extending it with functionalities for enabling the composition of Web services (Figure 3.2). Through WSDL, a remote service understands how to interact with a Web service. Through WSTL, a remote service knows the transaction support of the Web service. Besides the description of the transaction behavior of Web services, WSTL is also used to specify all other mediator related tasks such as: the

specification of mapping information for resolving representation and content dissimilarities, the definition of mediator service interfaces, and the specification of transactional interaction patterns of Web services compositions.

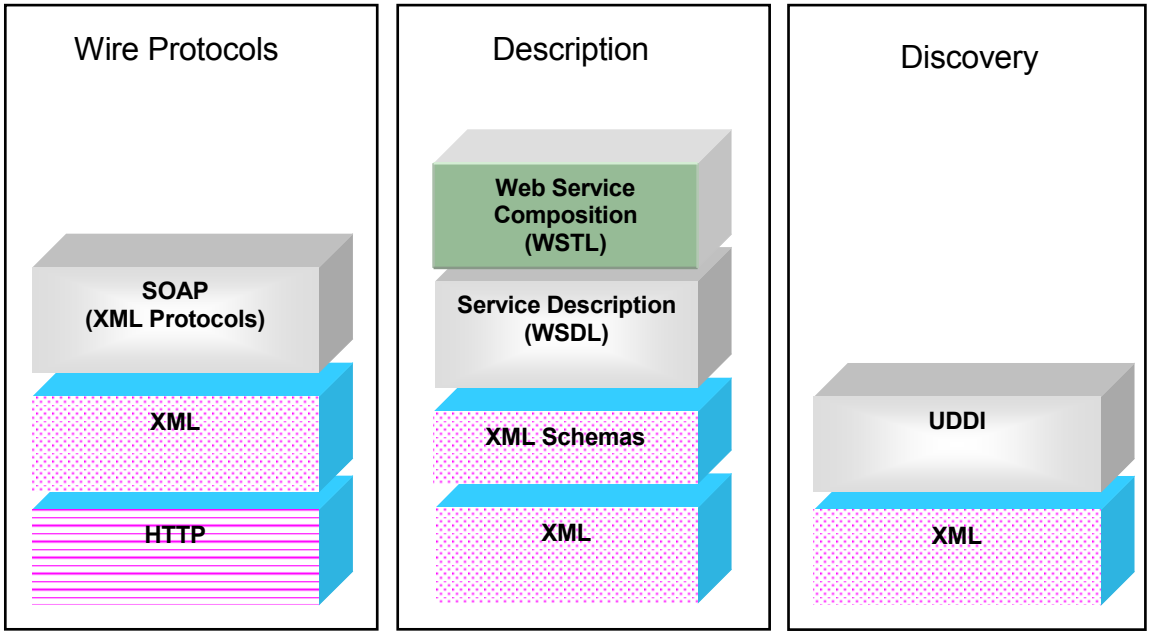


Figure 3.2 - The relation between WSTL and the XML-based technologies for enabling Web services.

The distributed architecture of WebTransact separates the task of aggregating and homogenizing heterogeneous Web services from the task of specifying transaction interaction patterns, thus providing a general mechanism to deal with the complexity introduced by a large number of Web services.

The design of the WebTransact framework provides novel special features for dealing with the problems of Web services composition. Since mediator services provide a homogenized view of Web services, the composition developer does not have to deal with the heterogeneity nor the distribution of Web services. Moreover, composition specification does not directly references Web service interfaces, consequently changes in the latter does not affect the former. In an environment driven by autonomy, this is an essential feature. Another import aspect of the WebTransact architecture is the explicit definition of transaction semantics. Since Web services describe their transaction support through WSTL definition, reliable interaction patterns can be specified through mediator service compositions.

The next section briefly describes the components of the WebTransact architecture.

3.1.1 The Remote Service Layer

Each remote service is a logical unit of work that performs a set of *remote operations* at a particular site. Besides its signature, a remote operation has a well-defined *transaction behavior*, which can be: *compensable*, *retriable*, or *pivot*, as in the Flex Transaction model [33]. A remote operation is compensable if, after its execution, its effects can be undone by the execution of another remote operation. Therefore, for each compensable remote operation, it must be specified which remote operation has to be executed in order to undo its effects. A remote operation is retriable, if it is guaranteed that it will succeed after a finite set of repeated executions. A remote operation is pivot, if it is neither retriable nor compensable. In WebTransact, the concept of compensable and pivot service is used to guarantee safe termination of compositions. Recall that remote service providers may not provide transaction functionalities like a two-phase commit interface. In this case, it is not possible to put a remote operation in a wait state like the prepared-to-commit state. Therefore, after a pivot service has been executed, its effects are made persistent and, as there is no compensating service for pivot services, it is not possible to compensate its persistent effects. For that reason, pivot remote operations can be executed only when, after its execution, the composition reaches a state where it is ready to successfully commit. At this state, the system has guaranteed that there will be no need to undo any already committed service.

Figure 3.3 shows the state transition diagrams for each type of remote operation. In the *pivot* remote operations, only the state transition between states not-executed and executing is controlled by the WebTransact system, the other transitions are controlled by the underlying system responsible for the execution of the remote operation. Therefore, after a pivot remote operation reaches the executing state, the WebTransact system has no control over the future states of the remote operation. In the *retriable* remote operations, the WebTransact system has control over transition between states not-executed and submitted, and also between states submitted and executing. The successful commit of the remote operation is assured, since the transition between states submitted and executing is controlled by the WebTransact system. The state transition diagram of *compensable* remote operations is similar to the state transition diagram of pivot remote operations, except for the transition between states committed and compensated. This transition is

controlled by the mediator service, so that it is possible to undo the effects of the remote operation, case the need arise. Figure 3.3 shows one more type of remote operation, the *virtual-compensable* remote operation. This type represents all remote operations that support the standard two-phase commit protocol (2PC). These services are treated like compensable services, but, actually, their effects are not compensated by the execution of another service, instead, they wait in the prepare-to-commit state until the composition reaches a state in which it is safe to commit the remote operation. Therefore, virtual-compensable remote operations references Web service operations whose underlying system provides (and exposes) some sort of distributed transaction coordination.

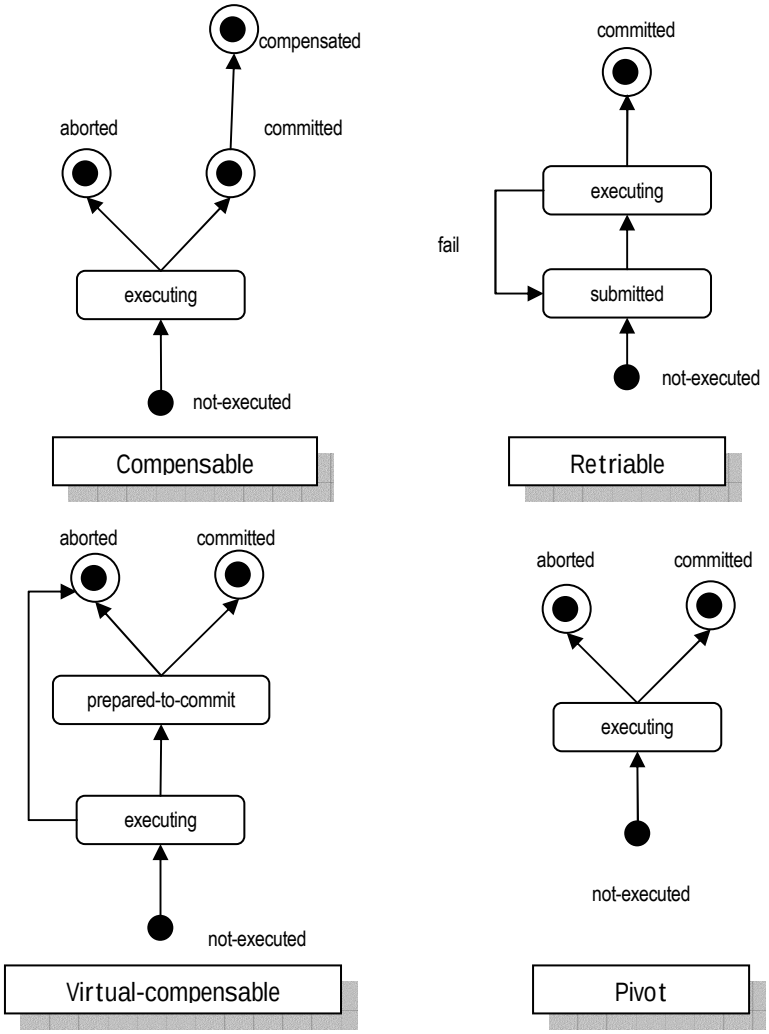


Figure 3.3 - Diagrams of the execution state transitions of Remote Service operations.

3.1.2 The Mediator Service Layer

Mediator services aggregate semantically equivalent remote services, thus providing a homogenized view of heterogeneous remote services. Unlike remote services, which are logical units of work that perform remote operations at a particular site, mediator services are *virtual* services responsible for delegating its operations' execution to one or more remote services. This delegation is done over a set of semantically equivalent remote services aggregated by the mediator service. Like remote operations, mediator service operations have a signature and a well-defined *transaction behavior*, which can be either *compensable*, *retriable*, or *pivot*.

The transaction behavior of one mediator service operation is based on the transaction behavior of its aggregated remote operations. If all aggregated remote operations have the same type of transaction behavior, e.g., *compensable*, then the transaction behavior of mediator service operation will have the same value, i.e., *compensable*. On the other hand, if the mediator service operation aggregates remote operations with different transaction behaviors, then its transaction behavior will be the *least restrictive* transaction behavior among the transaction behaviors of its aggregated remote operations (Figure 3.4). The most restrictive transaction behavior is the pivot transaction behavior, while the least restrictive is the compensable transaction behavior. This ranking is used to guarantee the safe termination of compositions. A mediator service operation, which aggregates at least one remote operation that is compensable, can participate in any composition execution. On the other hand, a mediator service operation that aggregates only pivot remote operations can participate only in compositions that call this mediator service operation after it reaches a state where it is ready to successfully commit. Recall from Section 3.1.1 that after a pivot remote operation has been executed, the composition has to enter a state where it is ready to successfully terminate. Thus, it is guaranteed that there will be no need to undo any already submitted pivot remote operation. Therefore, mediator service operations that aggregate only pivot remote service operations can only participate in compositions that call this mediator service operation when it is ready to successfully terminate.

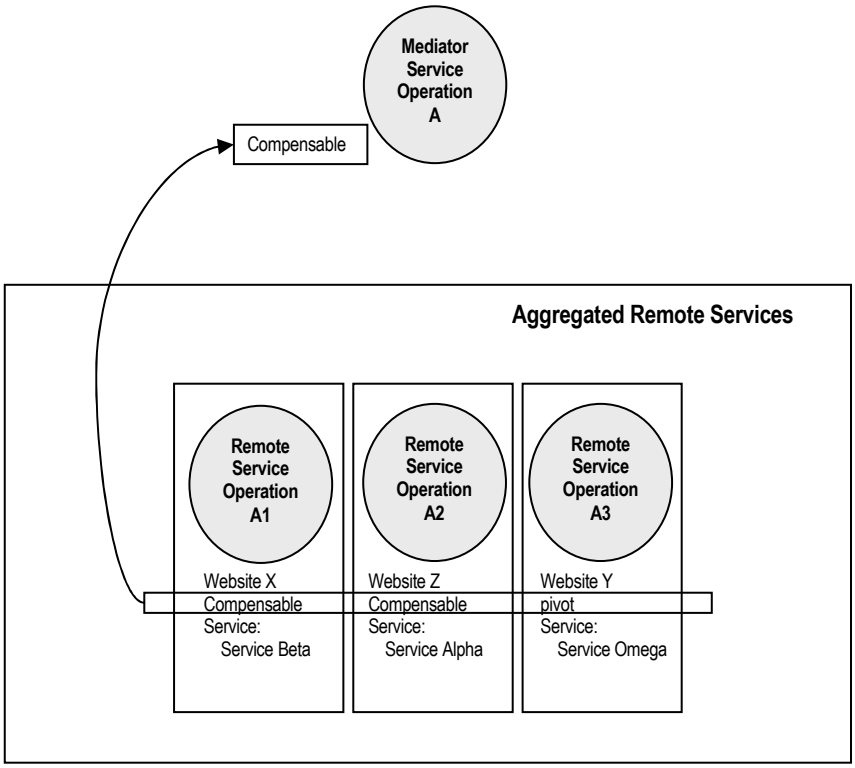


Figure 3.4 - Example of the transaction behavior of mediator services.

Each type of mediator service operation has the same state transition diagram of its related type of remote operation (Figure 3.3), except for the absence of the virtual-compensable operation. The specific transaction behavior of virtual-compensable remote operations is isolated by the mediator service operation that aggregates them. Mediator service operations that aggregate virtual-compensable remote operations expose the same interface of mediator service operations that aggregate only real compensable remote operations. This simplifies the protocol that manages the composition execution, as it is described next.

3.1.3 Resolving Semantic and Content Dissimilarities of Web Services

In order to provide a homogenized layer of services, each mediator service exposes a single interface that is used by composition specifications. As mediator services aggregate semantically equivalent remote services, which possibly have different interfaces, it is necessary to provide mapping information between the interface supported by the mediator service and each one of the interfaces supported by its aggregated remote services. In WebTransact, each WSDL port type is imported as a new remote service. Since the WSDL port type element defines the syntax for calling a set of remote

operations, i.e., a specific supported interface, each WSDL port type definition is considered as a separated remote service. A remote service links a mediator service to a WSDL port type element and it provides mapping information between mediator service operations and port type operations and specifies the content description of the Web service represented by that port type element. The mapping information prescribes how the input parameters of a remote service operation are constructed from the input parameters of its related mediator service operation, as well as how the output parameters or fault messages received from that remote service operation are mapped to the output or fault messages of its related mediator service operation. The content description specifies whether a remote service is able to execute a particular service invocation. For example, consider remote services rm_1 and rm_2 providing car reservations. Remote service rm_1 can make car reservations world wide, while remote service rm_2 accepts only car reservations in Brazil. Now, consider that mediator service ms1 aggregates rm_1 and rm_2. If ms_1 receives a request to make a car reservation inside USA then, ms_1 will invoke only the remote service rm_1. The mediator service ms_1 knows, using the remote service content description, that rm_2 is not able to make car reservation outside Brazil.

3.1.4 The Composite Mediator Service Layer

A composite mediator service describes transaction interaction patterns from a set of cooperating mediator service operations necessary to accomplish a task. Such interaction pattern defines the execution sequence of mediator service operations as well as the level of atomicity and reliability of a given composition. In WebTransact, a composition is specified using WSTL elements.

WSTL models compositions as *composite tasks*. A composite task is represented by a labeled directed graph in which nodes represent steps of execution and edges represent the flow of control and data among different steps. Each step of a composite task is either an atomic *task* or another composite task. An atomic task is a unit of work that is executed by a mediator service operation. Therefore, atomic tasks have a mediator service operation assigned to it, which is invoked when the task is executed .

Tasks are identified by a name and have a signature, a set of execution dependencies, a set of data links, and, optionally, a set of rules.

The *signature* of an atomic task is related to the input, output, and fault messages of the mediator service operation that is used as the implementation of the task. The *signature* of a composite task is related to the input, output, and fault messages of the component tasks used as the implementation of the task. A component task can be an atomic task or another composite task.

Execution dependencies are based on the execution state of component tasks defining the first kind of edges in the graph that represent a composite task. An execution dependency is defined among related component tasks and it is a constraint on the temporal occurrence of the start and termination events of them. Execution dependencies define the order in which tasks must be executed, i.e., the composition control flow. An execution dependency is always specified based on the *execution state* of one component task.

Data links are mappings between messages belonging to the signatures of related tasks to allow the exchange of information between these tasks. Data links are the second kind of edges in the graph that represents a composite task.

Rules specify the conditions under which certain events will happen. Rules can be associated to dependencies or to data links. A dependency that has a rule will evaluate to true if both the dependency *and* the rule evaluate to true. A data link that has a rule will evaluate to true if the rule evaluates to true. Data links without explicit rules always evaluate to true.

Besides the components described above, composite tasks have a set of *mandatory tasks*. This set specifies the component tasks that must commit in order to commit the composite task. A user can specify a composite task aggregating tasks that are desirable, but not essential, to accomplish the target task. The set of mandatory tasks allow the distinction between *desirable* and *mandatory* tasks, providing more flexibility while specifying a composition. This flexibility increases the composition robustness, since the set of tasks required to commit, in order to commit the composition, are formed only by that tasks that are essential to accomplish the composition target. Thus, the composition will successfully terminate even if a subset of its component tasks fails, as long as all its mandatory tasks successfully commit.

3.2 EXECUTION MODEL

In this section, we just sketch the operational semantics of WebTransact. In Chapter 7, we present the details of the transactional model of WebTransact, including its formal description and its protocols.

The execution model of WebTransact has two levels of control. The first is the composite task coordination level and the second is the atomic task coordination level. The first level specifies the rules for interpreting a composite task specification. Such rules ensure that a given composite task will be executed according to the semantics of its specification and guarantees that such execution will run as a single transaction. The second level specifies the rules for coordinating the execution of the mediator service operations that implement atomic tasks. This level is responsible for providing a transactional interface for all atomic tasks, which is used by the first level for coordinating composite task executions. The coordination of atomic tasks deals with problems that arise due to the one-to-many relationship that exists between mediator service operations - that implement atomic tasks - and remote service operations - that are the concrete service providers, as well as with the problems that arise due to the dissimilar transaction behavior of semantically equivalent remote service operations.

The execution of a composite task consists of two different phases: the *verification phase*, and the *scheduling phase*. The goal of the verification phase is to ensure the *guaranteed-termination* property of a given composite task instance. Since atomic tasks can be implemented by no-compensable mediator service operations, it is necessary to verify whether a composite task instance can be executed without leaving any side effects. In the verification phase, all component tasks are queried about their supported transaction behavior in order to analyze the structure of the resulting task execution. If the resulting structure conforms to certain properties then the instance is safe, i.e., it ensures the guaranteed-termination property, and can be scheduled to run. Otherwise, the instance is not safe and the WebTransact system returns an error message to the caller program. Instances of composite tasks that have passed through the verification phase are the input data of the scheduling phase. In this phase, the WebTransact schedules the component tasks of a composite task instance according to their execution dependencies.

The execution model of atomic tasks defines how an invocation of a given abstract mediator service operation is, in fact, realized by concrete remote service providers.

Atomic task instances are submitted for execution during the scheduling phase of composite tasks. Such event triggers the invocation of the mediator service operation that implements that atomic task. Since the logic of a mediator service operation is implemented by its aggregated remote service operations, when such operation is invoked, it is necessary to specify how its aggregated remote service operations should be used to attend that invocation.

4. Expressing Transaction Behavior of Web Services with WSTL

In Section 2.5 we have seen that WSDL describes the abstract operations as well as the application specific protocol supported by a particular Web service. However, to incorporate a Web service into a transactional Web service composition such information is not sufficient, it is also necessary to know the transaction behavior of each operation supported by the Web service. Therefore, it is necessary to extend WSDL with new elements capable of describing the transaction behavior of Web service operations. To accomplish that, we propose the "Web Services Transaction Language" (WSTL).

WSTL introduces new elements to describe Web services. However, it is not an independent type system; rather it extends WSDL in a dependent way. WSTL allows existent Web service descriptions to be extended, without modifications in the original WSDL description, in order to express the transaction semantics of its operations. WSTL is an XML language that supports the XML Schema specification (XSD) ([141], [142]) as its canonical type system and makes use of the extensibility mechanism provided by WSDL in order to incorporate the necessary type definitions for the description of the transaction behavior of Web services.

The WSDL framework allows inserting elements that represent a specific technology as child elements of its own elements. These elements are referred to as *extensibility elements*. In WSDL, extensibility elements are commonly used to specify the binding for some specific technology. The elements defined in WSTL are incorporated in WSDL through extensibility elements. However, rather than defining new technology binding elements, WSTL provides new abstract elements that specify the transaction behavior of WSDL operations.

The WSTL definitions described in this chapter enable the coordination of transactional executions of service compositions constituted by Web services with dissimilar transaction behavior. These definitions are used by the transaction model of WebTransact to guarantee the safe and correct execution of Web services compositions (Chapter 7). However, the WSTL elements for specifying transaction support of Web

services can be used by other systems that need to provide reliability in the Web services environment.

The next section introduces the notation conventions used for describing the WSTL framework. The WSTL elements that describe the Web service transaction behavior are described in detail along Section 4.2.

4.1 NOTATIONAL CONVENTIONS

1) The following namespace prefixes are used throughout this chapter:

Prefix	namespace URI	definition
wstl	http://schemas.xmlsoap.org/wsdl/wstl/	WSTL namespace for WSTL framework.
tmmap	http://schemas.xmlsoap.org/wsdl/tmmap/	WSTL namespace for Transaction Manager Mapping.
wsdl	http://schemas.xmlsoap.org/wsdl/	WSDL namespace as defined by WSDL.
soap	http://schemas.xmlsoap.org/wsdl/soap/	SOAP binding namespace as defined by WSDL.
http	http://schemas.xmlsoap.org/wsdl/http/	HTTP GET & POST binding namespace
mime	http://schemas.xmlsoap.org/wsdl/mime/	MIME binding namespace as defined by WSDL.
soapenc	http://schemas.xmlsoap.org/soap/encoding/	Encoding namespace as defined by SOAP 1.2 [144].
soapenv	http://schemas.xmlsoap.org/soap/envelope/	Envelope namespace as defined by SOAP 1.2 [144].
xsi	http://www.w3.org/2000/10/XMLSchema-instance	Instance namespace as defined by XSD .
xsd	http://www.w3.org/2000/10/XMLSchema	Schema namespace as defined by XSD [141].

tns	(various)	The "this namespace" (tns) prefix is used as a convention to refer to the current document.
(other)	(various)	All other namespace prefixes are samples only. In particular, URIs starting with "http://example.com" represent some application-dependent or context-dependent URI [12].

2) This work uses the same informal syntax used in the WSDL specification to describe the XML grammar as follows:

   a) The syntax appears as an XML instance, but the values indicate the data types instead of values (meta-language).

   b) Characters are appended to elements and attributes as follows: `"?"` (0 or 1 elements should appear), `"*"` (0 or more elements should appear), `"+"` (1 or more elements should appear).

   c) Elements bounded by <!—choice --> indicates that only one of the bounded elements should appear in the document.

   d) Element names ending in "…" (such as `<element…/>` or `<element…>`) indicate that elements/attributes irrelevant to the context are being omitted.

   e) The XML namespace prefixes (defined above) are used to indicate the namespace of the element being defined.

   f) Examples starting with `"<?xml"` contain enough information to conform to WSTL specification; other examples are fragments and require additional information to be specified in order to conform.

4.2 EXPRESSING TRANSACTION BEHAVIOR WITH WSTL

WSTL introduces the *transaction behavior* concept for defining the transaction behavior of Web services. For each operation of a WSDL document, there must be one WSTL definition describing its transaction behavior. The transaction behavior indicates the transaction semantics supported by that operation, i.e., whether the operation is *compensable, virtual-compensable, pivot, or retriable.*

The transaction behavior of compensable operations can have two different semantics: the *active action* semantics, and the *passive action* semantics. The active action semantics represents compensable operations that require the execution of other operation to compensate its effects. Operations that follow this semantics must designate which is(are) the compensatory operation(s) that can be used to compensate their effects, as well as the necessary mapping information to allow the building of the input message such compensatory operation(s). The passive action semantics represents compensable operations that do not require any external action to compensate their effects.

The transaction behavior semantics of virtual-compensable operations is used to represent Web service operations whose underlying systems expose some sort of transaction support on the Web.

The transaction behavior semantics of retriable operations is used to represent Web service operations that are guaranteed to successfully commit after a finite number of tries. The transaction behavior semantics of pivot operations is used to represent Web service operations are neither retriable nor (virtual-)compensable. Retriable operations are also non-compensable. In the next sections, we describe the WSTL elements that implement the transaction behavior concept.

4.2.1 Relationship between WSTL and WSDL definitions

WSDL allows extensibility elements inside many WSDL elements. In the next paragraphs, we relate WSDL to WSTL.

Considering the WSDL concept of abstract and concrete definitions, the transaction behavior describes the transaction semantics of *abstract operations* of Web services**.** The transaction behavior is a semantic concept related to operations. Thus, it is independent of network deployment or data format bindings of concrete endpoints and messages.

Therefore, the transaction behavior should be inserted as a child element of a port type operation that describes the abstract portion of message exchanges. However, WSDL does not allow extensibility elements inside port type operations. The only WSDL element that supports extensibility and is located in the context of the abstract portion of a WSDL document is the definitions element. For this reason, WSTL defines its root element, `transactionDefinitions`, as a direct child of the `wsdl:definitions` element (Figure 4.1).

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>

  <import namespace="uri" location="uri"/>*

  <wsdl:documentation…/>?

  <--    Types Element Definitions    -->*
  <--    Messages Element Definitions        -->*
  <--    PorTypes Element Definitions        -->*
  <--    Bindings Element Definitions        -->*
  <--    Services Element Definitions        -->*

  <wstl:transactionsDefinitions>?
    <wsdl:documentation …/>?

    <wstl:tmElem name=”nmtoken”>*
      <wsdl:documentation …/>?
    </wstl:tmElem>

    <wstl:trasactionBehavior name=”nmtoken” … >*
      <wsdl:documentation …/>?

      <!—choice -->
        <wstl:activeAction name=”nmtoken” … >*
          <wsdl:documentation …/>?
        </wstl:activeAction>

        <wstl:passiveAction name=”nmtoken” … >?
          <wsdl:documentation …/>?
        </wstl:passiveAction>

        <wstl:tmSrv>?
          <wsdl:documentation …/>?

          <wstl:tmRef tmElemName=”qname”>+
            <wsdl:documentation …/>?
          </wstl:tmRef>

          <wstl:tmElem name=”nmtoken”>+
            <wsdl:documentation …/>?
          </wstl:tmElem>

        </wstl:tmSrv>
      <!—choice -->

    </wstl:trasactionBehavior>

  </wstl: transactionsDefinitions>
</wsdl:definitions>
```

Figure 4.1 - WSTL elements embedded within WSDL document.

The `transactionDefinitions` element has two child elements: the `tmElem` element, which contains information on transaction managers to be used by virtual-

compensable operations; and the `transactionBehavior` element, which contains information on the transaction semantics of service operations.

The `transactionbehavior` is defined using three main elements:

1) `activeAction`, which specifies compensatory operations for compensable operations, as well as the necessary mappings among the compensable operation parameters and the input parameters of its related compensatory operation.

2) `passiveAction`, which specifies information on compensable operations that do not require any external action to compensate its effects.

3) `tmSrv`, is used by virtual-compensable Web service operations to designate the transaction manager that must be contacted in order to involve such operation in the coordination of a distributed transaction.

These elements are described in depth in the next sections. The transaction definitions element is described in Section 4.2.4. The transaction behavior element is described in Section 4.2.5. The active action element is presented in Section 4.2.6. Next, Section 4.2.7 describes the passive action element. Finally, the `tmElem` and the `tmSrv` elements are defined in Section 4.2.9.

Next, we present a WSTL document example that will be used to highlight language features and how WSTL elements fit the WSDL framework.

4.2.2 WSTL document Example

The following example (Figure 4.2, Figure 4.3, Figure 4.3, and Figure 4.4) shows a WSDL definition of a simple service providing car reservations, extended by the WSTL framework. The car reservation service supports two operations: `reservation` and `cancelReservation`.

The `reservation` operation has two groups of parameters, one related to pickup information, called `pickupInfo`; and the other related to drop-off information, called `dropoffInfo`. Both the `pickupInfo` and the `dropoffInfo` parameters are of type `infoType`, which has three simple fields: `dt`, for date, of type `dateTime`; `location` of

type `string`; and `time` of type `integer`. The `reservation` operation returns a reservation code, or an error, in the parameter `reservationResult` of type `string`.

The `cancelReservation` operation has only one parameter: `reservationCode` of type `string`. A valid value for this parameter is the value returned in a previous successfully executed request of operation `reservation`. The `cancelReservation` operation returns a value of type `string` indicating the success or failure of the executed request.

Both the `reservation` and `cancelReservation` operations are deployed using the SOAP 1.1 protocol over HTTP.

In this example, we make use of the `import` WSDL element [137] that allows the separation of different elements of a service definition into independent documents. The use of the `import` element improves reusability of WSDL definitions and also helps writing neatened service definitions, by separating the definitions in their level of abstraction, that are easier to maintain. Besides these features, the `import` element also makes it easy to aggregate new information to existent definitions as shown in the example. The definition of the car reservation service is separated in four documents: data type definitions (Figure 4.2), abstract definitions (Figure 4.3), service bindings (Figure 4.3), and transaction behavior definitions (Figure 4.4). Appendix 1 shows the same service definition without the use of the import element.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://example.com/carReservation/Schema/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns="http://example.com/carReservation/Schema/"
  <types>
    <xsd:schema attributeFormDefault="qualified"
                elementFormDefault="qualified"
                targetNamespace="http://example.com/carReservation/Schema/">

      <xsd:element name="reservation">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="pInput" type="tns:reservationInputType"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:complexType name="reservationInputType">
        <xsd:sequence>
          <xsd:element name="pickupInfo" type="tns:infoType"/>
          <xsd:element name="dropoffInfo" type="tns:infoType"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="infoType">
        <xsd:sequence>
          <xsd:element name="dt" type="xsd:dateTime"/>
          <xsd:element name="location" type="xsd:string"/>
          <xsd:element name="time" type="xsd:int"/>
```

```
      </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="reservationResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="reservationResult" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="cancelReservation">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="reservationCode" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="cancelReservationResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="cancelReservationResult" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="string" type="xsd:string"/>
  </xsd:schema>
 </types>
</definitions>
```

Figure 4.2 - Types definition for car reservation service - file: "http://example.com/carReservation/carReservation.xsd".

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://example.com/carReservation/abstractDef/"
  xmlns:tns="http://example.com/carReservation/abstractDef/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:lxsd="http://example.com/carReservation/Schema/">

  <import namespace="http://example.com/carReservation/Schema/"
          location="http://example.com/carReservation/carReservation.xsd"/>

  <message name="reservationSoapIn">
    <part name="parameters" element="lxsd:reservation"/>
  </message>
  <message name="reservationSoapOut">
    <part name="parameters" element="lxsd:reservationResponse"/>
  </message>
  <message name="cancelReservationSoapIn">
    <part name="parameters" element="lxsd:cancelReservation"/>
  </message>
  <message name="cancelReservationSoapOut">
    <part name="parameters" element="lxsd:cancelReservationResponse"/>
  </message>
  <portType name="reservationSoap">
    <operation name="reservation">
      <input message="tns:reservationSoapIn"/>
      <output message="tns:reservationSoapOut"/>
    </operation>
    <operation name="cancelReservation">
      <input message="tns:cancelReservationSoapIn"/>
      <output message="tns:cancelReservationSoapOut"/>
    </operation>
  </portType>
</definitions>
```

Figure 4.3 - Abstract definitions for car reservation service - file:"http://example.com/carReservation/carReservationAbsDef.wsdl"

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://example.com/carReservation/services/"
```

```
  xmlns:tns="http://example.com/carReservation/services/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:absd="http://example.com/carReservation/abstractDef/”
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">

  <import namespace=”http://example.com/carReservation/abstractDef/”
           location=”http://example.com/carReservation/
                carReservationAbsDef.wsdl”/>

  <binding name="reservationSoap" type="absd:reservationSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                 style="document"/>
    <operation name="absd:reservation">
      <soap:operation soapAction="http://example.com/reservation"
                   style="document"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
    <operation name="absd:cancelReservation">
      <soap:operation
          soapAction="http://example.com/cancelReservation" style="document"/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="carReservation">
    <port name="reservationSoap" binding="tns:reservationSoap">
      <soap:address location="http://example.com/carReservation/"/>
    </port>
  </service>
</definitions>
```

Figure 4.4 - Binding and service definitions for car reservation service - file “http://example.com/carReservationSvc.wsdl”.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
  targetNamespace="http://example.com/carReservation/transacBehavior/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wstl="http://schemas.xmlsoap.org/wsdl/wstl/"
  xmlns:absd="http://example.com/carReservation/abstractDef/””>

  <import namespace=”http://example.com/carReservation/abstractDef/”
           location=”http://example.com/carReservation/
                carReservationAbsDef.wstl”/>

  <wstl:transactionDefinitions>
    <wstl:transactionBehavior operationName=" absd:reservation"
                    type="compensable">
      <wstl:activeAction portTypeName="svc:reservationSoap"
                  compensatoryOper="cancelReservation">
        <wstl:paramLink>
          <wstl:sourceParamLink msgName="reservationSoapOut"
                param="absd:reservationResponse/@reservationResult"/>
          <wstl:targetParamLink msgName="cancelReservationSoapIn"
                param="absd:cancelReservation/@reservationCode"/>
        </wstl:paramLink>
      </wstl:activeAction>
    </wstl:transactionBehavior>
    <wstl:transactionBehavior operationName=" absd:cancelReservation"
                    type="retriable"/>
  </wstl:transactionDefinitions>
```

```
</definitions>
```

Figure 4.5 - Transaction behavior definitions for car reservation service - file "http://example.com/carReservation.wstl".

4.2.3 Graphical Notation for WSTL elements

In order to facilitate the readability of the XML schemas' elements that will be presented in the next sections, we use a simple graphical representation that is sketched in Appendix 4.

4.2.4 TransactionDefinitions Element

The root element of the WSTL framework is the `transactionDefinitions` element. This element is appended to a WSDL document as an extensibility element under the WSDL `definitions` element.

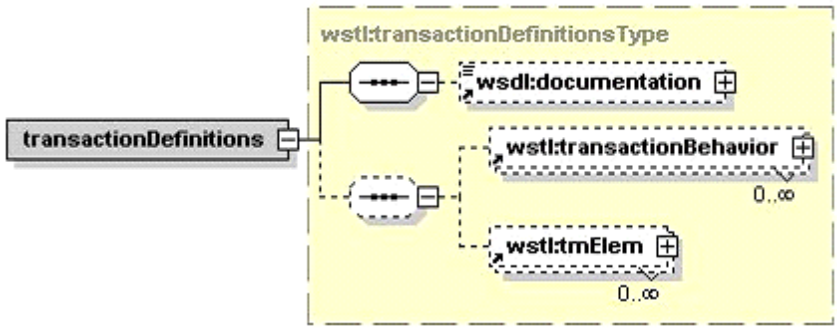


Figure 4.6 - WSTL `transactionDefinitions` element

The `transactionDefinitions` element consists of zero or one `documentation` element, zero or more `transactionBehavior` elements, and zero or more `tmElem` elements. All WSTL elements have an optional `wsdl:documentation` element as a container for human readable documentation. As defined in the WSDL specification [137], the content of this element is arbitrary text and elements. Both the set of `transactionBehavior` and `tmElem` elements under the `transactionDefinitions` element are optional with the intention of allowing the definition of these elements in separate documents. Figure 4.7 shows the XML Schema fragment of the `transactionDefinitions` element.

```
<xsd:element   name="transactionDefinitions"
               type="wstl:transactionDefinitionsType">
  <xsd:complexType name="transactionDefinitionsType">
    <xsd:complexContent>
```

```
    <xsd:extension base="wsdl:documented">
      <xsd:sequence minOccurs="0">
        <xsd:element ref="wstl:transactionBehavior"
                 minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="wstl:tmElem" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 4.7 - XML schema fragment for `transactionDefinition` element and `transactionDefinitionType` type.

The goal of the set of `transactionBehavior` elements is to describe the transaction semantics of each operation of a given Web service. The goal of the set of `tmElem` elements is to describe information on transaction managers that can be used to manage transactions of virtual-compensable operations.

In the car reservation service (Figure 4.), the transactionDefinitions element has two child `transactionBehavior` elements each containing transaction semantics information on the operations supported by that Web service. In the next section, we describe the `transactionBehavior` element and the `tmElem` element is described in Section 4.2.8.

4.2.5 TransactionBehavior Element

The `transactionbehavior` element contains information on the transaction behavior of a given service operation, according to the semantics defined in Section 3.1.1.

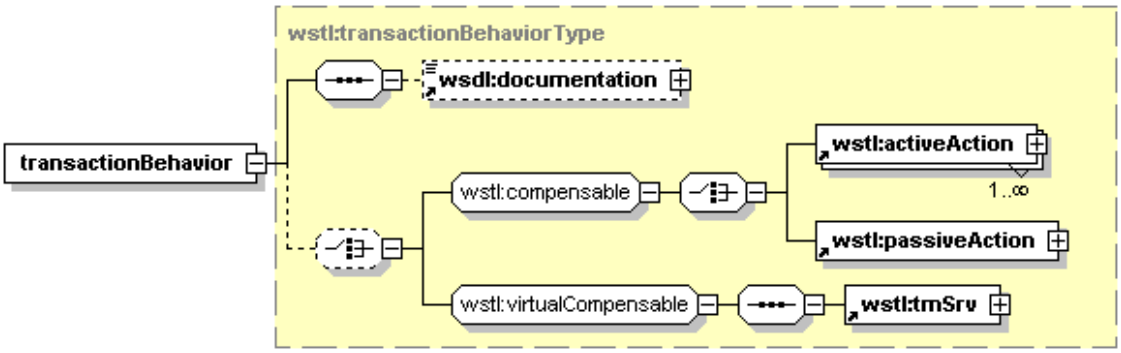


Figure 4.8 - WSTL `transactionBehavior` element.

This element has three attributes: `name`, `type`, and `operationName` (Figure 4.9). The `name` attribute provides a unique name among all `transactionBehavior` elements within the enclosing WSTL document. The `operationName` attribute links the

transactionBehavior element to a particular wsdl:operation. WSTL follows the linking rules of WSDL framework [137]. All the references to a WSTL or WSDL element are made using a QNAME [137]. The resolution of QNAMES in WSDL is similar to the resolution of QNAMES described in the XML Schemas specification [141]. The type attribute indicates the transaction behavior type of the wsdl:operation that is being referenced. Possible values for this attribute are pivot, retriable, compensable, and virtualCompensable, as defined by the transactionBehaviorEnum type (Figure 4.9).

The transactionBehavior element supports two types of child definitions[5] (Figure 4.8):

1) Compensable, which describes the actions that should be taken to compensate a compensable operation; and

2) VirtualCompensable, which indicates the transaction manager that should be used to manage transactions that will run on behalf of a virtual-compensable operation. This definition is discussed in Section 4.2.8.

The action described by a Compensable transaction behavior is a choice between two elements:

1) The ActiveAction element, which specifies compensatory actions for compensable operations. These actions are wsdl:operation elements that should be executed to compensate the effects of the operation referred by the operationName attribute of the transactionBehavior element.

2) The passiveAction element, which specifies information on compensable operations that do not require any external action to compensate its effects.

Figure 4.9 shows the XML Schema fragment elements and types described in this section.

```
<xsd:element name="transactionBehavior" type="wstl:transactionBehaviorType"/>

<xsd:complexType name="transactionBehaviorType">
 <xsd:complexContent>
  <xsd:extension base="wsdl:documented">
```

[5] Recall from Section 4.2 that pivot and retriable Web services do not require further definitions besides the transaction behavior element itself.

```
   <xsd:choice minOccurs="0">
    <xsd:group ref="wstl:compensable"/>
    <xsd:group ref="wstl:virtualCompensable"/>
   </xsd:choice>
   <xsd:attribute name="name" type="xsd:NMTOKEN" use="optional"/>
   <xsd:attribute name="type"
                 type="wstl:transactionBehaviorEnum" use="required"/>
   <xsd:attribute name="operationName" type="xsd:QName" use="required"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="transactionBehaviorEnum">
 <xsd:restriction base="xsd:string">
  <xsd:enumeration value="pivot"/>
  <xsd:enumeration value="compensable"/>
  <xsd:enumeration value="virtualCompensable"/>
  <xsd:enumeration value="retriable"/>
 </xsd:restriction>
</xsd:simpleType>

<xsd:group name="compensable">
  <xsd:choice>
    <xsd:element ref="wstl:activeAction" maxOccurs="unbounded"/>
    <xsd:element ref="wstl:passiveAction"/>
  </xsd:choice>
</xsd:group>
 <xsd:group name="virtualCompensable">
  <xsd:sequence>
    <xsd:element ref="wstl:tmSrv"/>
  </xsd:sequence>
</xsd:group>
```

Figure 4.9 - XML schema fragment for `transactionBehavior` element, `transactionBehaviorType` type, and `transactionBehaviorEnum` type.

In the car reservation service (Figure 4.), there are two `transactionBehavior` elements, one for each `wsdl:operation` defined in the WSDL document "carReservationAbsDef.wsdl" (Figure 4.3).

The first `transactionBehavior` element has "`absd:reservation`" as the value of attribute `operationName`. The prefix "`absd:`" references the namespace "`http://example.com/carReservation/abstractDef/`", which is the namespace for the abstract definitions of the car reservation service. In the example, the WSTL document has an `import` element that imports the abstract definitions of the car reservation service and associates these definitions to the namespace "`http://example.com/carReservation/abstractDef/`". The value "`absd:reservation`" is a `QNAME` that references the `wsdl:operation` element named `reservation`, which is defined in the WSDL document "`http://example.com/carReservation/carReservationAbsDef.wsdl`". Therefore, the first `transactionBehavior` element in Figure 4. defines the transaction behavior of the `reservation` operation of the car reservation service. The value

"compensable" of attribute type indicates that reservation operation is *compensable*, as defined in Section 3.1.1.

The second transactionBehavior element has "absd:cancelReservation" as the value of attribute operationName. This value indicates that the transactionBehavior element defines the transaction behavior of the cancelReservation operation of the car reservation service. The value "retriable" of attribute type indicates that the cancelReservation operation is *retriable*, as defined in Section 3.1.1. Note that there is no child element for this transactionBehavior element. The reason is that retriable operations are not compensable, thus there is no other necessary information on the operation transaction behavior to be provided by the Web service. Only *compensable* and *virtual-compensable* operations need further information besides the information available in the set of attributes of the transactionBehavior element.

4.2.6 ActiveAction Element

The activeAction element is used to describe the transaction behavior of compensable operation that requires a compensatory operation to undo its effects. This element contains the compensatory operation name and information on the values that should be used when invoking the compensatory operation. Recall that the compensable operation is defined in the operationName attribute of the transactionBehavior element that contains the activeAction element.

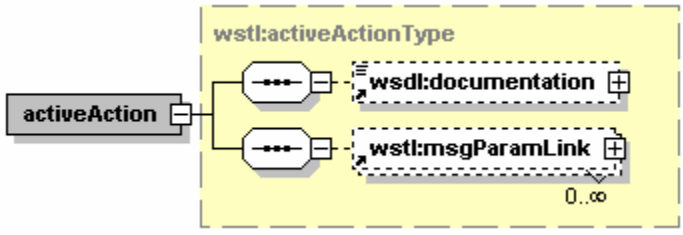


Figure 4.10 - WSTL activeAction element.

The activeAction element has three attributes: name, portTypeName, and compensatoryOper (Figure 4.11). The optional attribute name provides a unique name among all activeAction elements within the enclosing WSTL document. The mandatory portTypeName attribute refers to a wsdl:portType element using a

Qname. The mandatory compensatoryOper attribute refers to a wsdl:operation element using a Qname. The Qname used by both attributes follows the linking rules defined by WSDL [137]. The values of these attributes define a target compensatory operation for the operation selected in the transactionBehavior element. In addition to the portTypeName and compensatoryOper attributes, the activeAction element has an optional set of msgParamLink child elements (Figure 4.10) for specifying information on how input parameters of the target compensatory operation should be filled out. Next Section discuses the msgParamLink element in depth. Figure 4.11 shows the XML Schema fragment elements and types described in this section.

Note that a transactionBehavior element can have more than one activeAction element. Therefore, an abstract wsdl:operation can be compensated by more than one active action. The reason for this is that the same abstract wsdl:operation can belong to more than one wsdl:portType. Thus, if a wsdl:operation is a WSTL compensatory operation and it belongs to more than one wsdl:portType, then there will exist more then one wstl:activeAction element for that wsdl:operation, one for each existent wsdl:portType.

```
<xsd:complexType name="activeActionType">
 <xsd:complexContent>
  <xsd:extension base="wsdl:documented">
   <xsd:sequence>
    <xsd:element ref="wstl:paramLink" minOccurs="0"
                 maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:NMTOKEN" use="optional"/>
   <xsd:attribute name="portTypeName" type="xsd:QName" use="required"/>
   <xsd:attribute name="compensatoryOper" type="xsd:QName" use="required"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Figure 4.11 - XML schema fragment for activeAction element, and activeAction Type type.

In the car reservation service (Figure 4.), the first transactionBehavior element references the absd:reservation operation and indicates that it requires a compensatory operation. This compensatory operation is defined by the activeAction element that has absd:reservationSoap and absd:cancelReservation as the values of attributes portTypeName and compensatoryOper, respectively. These values define the cancelReservation operation of port type reservationSoap from the car reservation service (prefix absd:) as the compensatory operation for the

reservation operation. In this example, there is only one activeAction element because the compensatory operation absd:cancelReservation belongs to a single port type. Appendix 1 shows an example where there is one transaction behavior with more than one activeAction element.

4.2.6.1 MsgParamLink Element

The msgParamLink element represents a data link involving a message part of a compensable operation to a message part of its compensatory operation. This data link specifies a data flow from the compensable operation to its compensatory operation. The msgParamLink element allows WebTransact to infer which input values should be used when calling the compensatory operation.

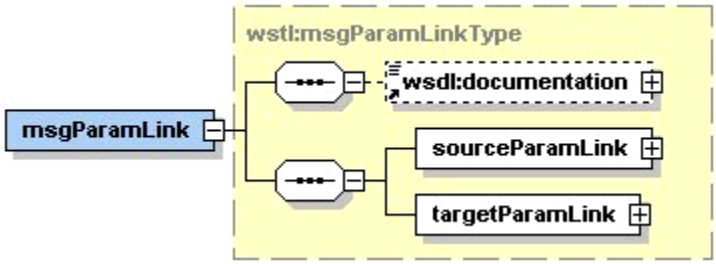


Figure 4.12 - WSTL msgParamLink element

The msgParamLink element contains the attribute name that provides a unique name among all msgParamLink elements within the enclosing WSTL document, and a set of two child elements: the sourceParamLink and targetParamLink elements (Figure 4.12). Each pair sourceParamLink and targetParamLink defines one data link between one input/output parameter of the compensable operation (represented by the sourceParamLink element) and one input parameter of the compensatory operation (represented by the targetParamLink element). Figure 4.13 shows the XML Schema fragment of the msgParamLinkType type.

```
<xsd:complexType name="msgParamLinkType">
 <xsd:complexContent>
  <xsd:extension base="wsdl:documented">
   <xsd:sequence>
    <xsd:element name="sourceParamLink" type="wstl:msgLinkType"/>
    <xsd:element name="targetParamLink" type="wstl:msgLinkType"/>
   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:NMTOKEN" use="optional"/>
  </xsd:extension>
 </xsd:complexContent>
```

```
</xsd:complexType>
```

Figure 4.13 - XML schema fragment for `msgParamLink` element, and `msgParamLinkType` type.

Both `sourceParamLink` and `targetParamLink` elements are of type `msgLinkType` (Figure 4.14). The `msgLinkType` type has three attributes: `name`, `msgName`, and `param`. The optional attribute `name` provides a unique name among all `sourceParamLink/targetParamLink` elements within the enclosing WSTL document. The mandatory `msgName` attribute refers to a `wsdl:message` element using a `Qname`. The `param` attribute of type `tmmap:XPathLinkType` identifies a parameter inside the `wsdl:message` referred by the `wstl:msgName` attribute. The `tmmap:XPathLinkType` type is explained in detail in the next section. Note that the namespace for the `XPathLinkType` type is not in the WSTL namespace. This utility type is defined in the `tmmap` namespace defined in the XML Schema of Appendix 6.

```
<xsd:complexType name="msgLinkType">
 <xsd:complexContent>
  <xsd:extension base="wsdl:documented">
   <xsd:attribute name="name" type="xsd:NMTOKEN" use="optional"/>
   <xsd:attribute name="msgName" type="xsd:QName" use="required"/>
   <xsd:attribute name="param" type="tmmap:XPathLinkType" use="required"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Figure 4.14 - XML schema fragment for `msgLinkType` type.

4.2.6.2 XPathLinkType Type

The `XPathLinkType` is a utility type defined as a subset of XPath expressions and used for defining data links in WSTL. Elements of `XPathLinkType` type must conform to the following BNF syntax:

[1] `XPathLinkType` ::= Path

[2] Path ::= (NameTest '/')* (NameTest | '@' NameTest)

[3] NameTest ::= QName | NCName ':'

This BNF syntax allows the selection of elements as well as attribute of elements defined in a WSDL document. Figure 4.15 shows the XML Schema fragment of the `XPathLinkType` type.

```
<xsd:simpleType name="XPathLinkType">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="(((\i\c*:)?(\i\c*))/)*(@((\i\c*:)?(\i\c*)))"/>
  </xsd:restriction>
</xsd:simpleType>
```

Figure 4.15 - The WSTL `XPathLinkType` type.

4.2.6.3 `MsgParamLink` Element Usage

To illustrate the usage of the `wstl:msgParamLink` element, first we describe a general example. In the next section, we show a concrete example using the car reservation service from Section 4.2.2.

Consider operation `A` and operation `B` in Figure 4.16. The input message `M1` of operation `A` has two parts called `P1` and `P2`. Message part `P1` is described by the simple date type `a`. Message part `P2` is described by a root complex type `b` consisting of simple string type `b2` and the complex type `b1.` The complex type `b1` consists of two simple types, the simple float type `b11` and simple string type `b12`. The output message `M2` of operation `A` has one part called `P3` described by the simple string type `c`. The input message `M3` of operation `B` has two parts called `P4` and `P5` described by the simple string type `d` and simple string type `e`.

If operation `B` is the compensatory operation for operation `A`, it is necessary to define two data links, one for each message part of operation `B`. In Figure 4.16 the dashed arrows named `dl1` and `dl2` represent these data links. In WSTL, the data link `dl1` is represented by a pair (`wstl:sourceParamLink`, `wstl:targetParamLink`) that links one message part of the compensable operation `A` to another input message part of compensatory operation `B`. A `sourceParamLink` element for representing data link `dl1` should have `M1` as its `msgName` attribute and the XPath [143] expression `b/b1/b12` as the value for its `param` attribute. The `msgNameAttribute` attribute sets the context node for the XPath expression, thus `M1` is the context node for the path `b/b1/b12`. A `targetParamLink` element for representing data link `dl1` should have `M3` as its `msgName` attribute and the XPath expression `e` as the value for its `param` attribute. The context node for the XPath expression `e` is `M3`. The data link `dl2` is represented by another pair (`wstl:sourceParamLink`, `wstl:targetParamLink`) in the same way as data link `dl1`. A `sourceParamLink` element for representing data link `dl2` should

have `M2` as its `msgName` attribute and the XPath expression `c` as the value for its `param` attribute. The context node for the XPath expression `c` is `M2`. A `targetParamLink` element for representing data link `dl2` should have `M3` as its `msgName` attribute and the XPath expression `d` as that value for its `param` attribute. The context node for the XPath expression `d` is `M3`.

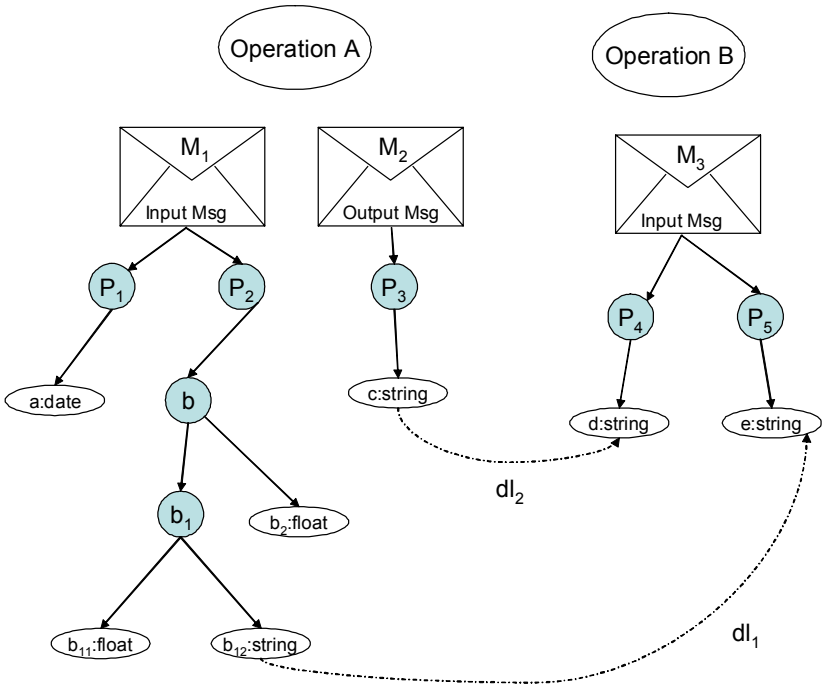


Figure 4.16 - Data link example.

4.2.6.4 The Car Reservation Example

Figure 4.17 shows a data link connecting operations of the car reservation service (Section 4.2.2). There is only one data link connecting the output message part of the reservation operation to the input message part of its compensating operation, cancel car reservation.

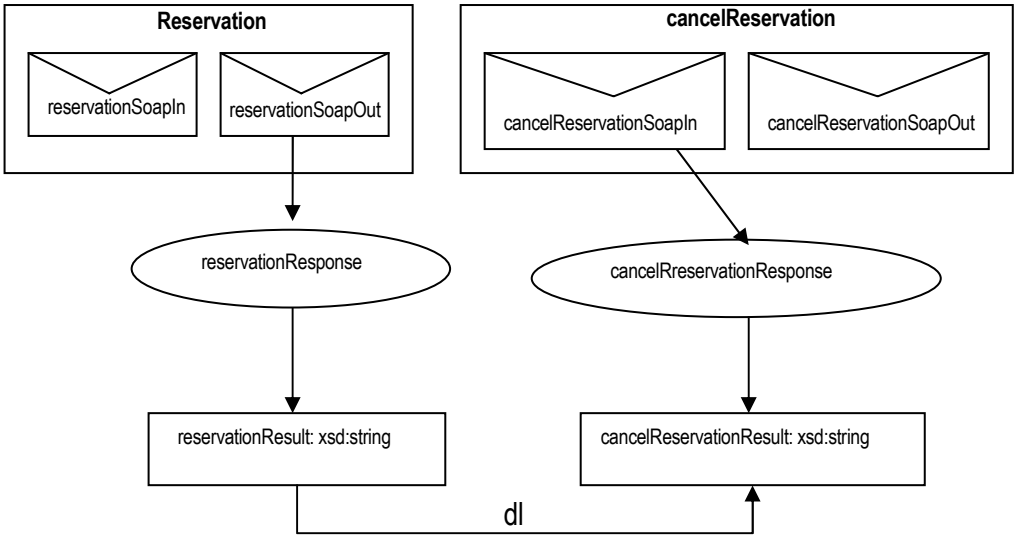


Figure 4.17 - Data link for the car reservation example.

The `sourceParamLink` element for representing this data link has `absd:reservationSoapOut` as its `msgName` attribute and the XPath expression `absd:reservationResponse/@reservationResult` as the value for its `param` attribute. Thus, context node for the XPath expression `absd:reservationResponse/@reservationResult` is `absd:reservationSoapOut`. The `targetParamLink` element has `absd:cancelReservationSoapIn` as its `msgName` attribute and the XPath expression `absd:cancelReservation/@reservationCode` as the value for its `param` attribute. The context node for the XPath expression `absd:cancelReservation/@reservationCode` is `absd:cancelReservationSoapIn`.

The data link defined above is used by WebTransact to construct the input message of the compensatory operation `cancelReservation` when invoking it to compensate the work done by the operation `reservation` during an execution of a WebTransact transaction.

4.2.7 PassiveAction Type

The `passiveAction` element is used to describe the transaction behavior of compensable operation that does not require a compensatory operation to undo its effects. This kind of compensable operation is not explicitly compensable because its effects are

not permanent, rather they have a well-defined period of life. For example, if a car reservation service requires a confirmation within a certain period of time, after which the reservation is discarded, then this reservation service should be described by a passiveAction element indicating when it will expire. Figure 4.18 shows the XML Schema fragment that defines the `passiveAction` element.

```
<xsd:element name=”passiveAction” type=”passiveActionType” >
<xsd:complexType name="passiveActionType">
 <xsd:complexContent>
  <xsd:extension base="wsdl:documented">
   <xsd:attribute name="name" type="xsd:NMTOKEN" use="optional"/>
   <xsd:attribute name="expiration" type="xsd:integer" use="required"/>
       <xsd:attribute name="unit" type="wstl:timeUnitType" use="required"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="timeUnitType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="minutes"/>
      <xsd:enumeration value="hours"/>
      <xsd:enumeration value="days"/>
      <xsd:enumeration value="weeks"/>
      <xsd:enumeration value="months"/>
      <xsd:enumeration value="year"/>
    </xsd:restriction>
</xsd:simpleType>
```

Figure 4.18 - XML schema fragment for `passiveAction` element, `passiveActionType` type, and `timeUnitType` simpleType.

4.2.8 Integrating Virtual-compensable Services in WebTransact

Recall from Section 3.1.1 that virtual-compensable services support the standard two-phase commit protocol (2PC) ([46], [72], [71]) and are treated by WebTransact as compensable services. However, the effects of a virtual-compensable service are not compensated by the execution of another service; instead, they wait in the prepare-to-commit state until the application reaches a state in which it is safe to commit the Web service. Therefore, virtual-compensable services must be supported by some kind of distributed transaction processing system that provides the standard two-phase commit protocol. There are many commercial distributed transaction processing systems currently in use ([86], [6], [117], [120], [62], [11]), which are based on different distributed transaction models ([96], [130], [87]). This feature of current distributed transaction systems raises interoperability issues in communicating and coordinating distributed transactions between WebTransact and other distributed transaction systems.

In the next section, we present how WebTransact addresses the interoperability issues of coordinating transactions spanning many different transactional distributed systems.

4.2.8.1 A Framework for Integrating Virtual-compensable Services

The WSTL framework defines a set of rules for integrating Web services that have their own local transaction support based on the two-phase commit protocol. The key concept of such integration is the use of the WSDL framework for exposing the 2PC communication interface of a given transaction manager.

The integration of virtual-compensable Web services can be summarized as follows. First, a transaction manager must expose its interface using the WSDL abstract concept of port types and then it must define the set of supported communication protocols through WSDL binding definitions. Next, the concrete address, or addresses, to reach that transaction manager must be defined as WSDL port elements. Finally, these ports must be combined to represent a transaction manager service. After these steps, a Web service can reference this transaction manager indicating that it is able to coordinate the Web service's local transaction processing.

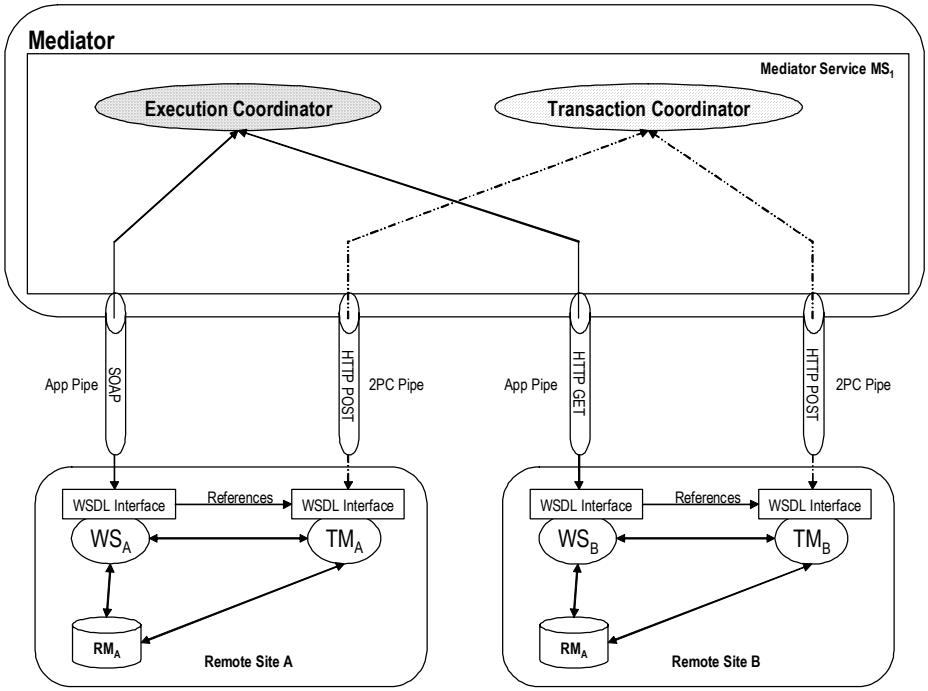


Figure 4.19 - Distributed Transaction Model for Virtual-compensable Web services.

Figure 4.19 shows the distributed transaction model for virtual-compensable Web services. The mediator service MS_1 aggregates two virtual-compensable Web services: WS_A at remote site A and WS_B at remote site B. At remote site A, the transaction manager TM_A is responsible for coordinating local transactions that involve Web service WS_A. The communication between mediator service MS_1 and remote site A is done following the two-pipe model [35]. As both Web service WS_A and transaction manager TM_A expose their operations through a WSDL interface, it is possible to define completely independent communication connections for each component. For example, if the WSDL document for WS_A defines a SOAP binding for its operations and the WSDL document for TM_A defines a HTTP POST binding for its operations then, the communication between mediator service MS_1 and Web service WS_A can be done using SOAP messages, while the communication between mediator service MS_1 and transaction manager TM_A can be done using HTTP POST messages. Likewise, depending on the WSDL definitions of Web service WS_B and transaction manager TM_B, the communication between mediator service MS_1 and Web service WS_B can be done using HTTP GET messages, while the communication between mediator service MS_1 and transaction manager TM_B can be done using HTTP POST messages.

It is important to note that exposing Web services and their transaction managers through WSDL interfaces allows virtually any communication protocol to be used when communicating these components with WebTransact. This is an important feature of the proposed distributed transaction model when comparing to the existent models ([86], [96], [127], [130]). The majority of the existent models employ a one-pipe model. That is, the transaction and application protocols are tightly integrated and execute over the same communication channel. When integrating virtual-compensable Web services, it is not reasonable to assume that only one communication channel is to be used. Web services are inherently autonomous and heterogeneous. Therefore, a platform for integrating such component has to accommodate as much diversity as possible in order to be effective. The proposed distributed transaction model exploits the features of the WSDL framework to allow the integration of any kind of Web service as well as their transaction manager systems. In the next section, we describe how WebTransact exploits the WSDL framework to integrate virtual-compensable services.

4.2.9 Defining Virtual-compensable Operations

Considering the distributed transaction model for Web services environments described in the previous section, three issues arise when coordinating distributed transactions between Mediator services and virtual-compensable Web services:

- For each virtual-compensable operation `op` of a given Web service `WS` there must be a related local transaction manager `TM` for providing the local transaction support for `op`.
- For each transaction manager `TM`, there must be a WSDL document for defining `TM`'s interface.
- The transaction manager of WebTransact must understand the underlying semantics of the interface exposed by each transaction manager `TM` in order to coordinate transactions involving `TM`.

The first issue is addressed by WSTL framework through the `VirtualCompensable` definition of the `wstl:transactionbehavior` element (Section 4.2.5). The `VirtualCompensable` definition of the `wstl:transactionbehavior` element indicates a set of transaction manager services that can be used to coordinate local transactions of a given operation. The second issue is addressed by the WSDL framework through the definition of a WSDL interface that exposes the 2PC interface of the remote transaction manager. The WSTL framework addresses the third issue defining a standard 2PC interface that must be supported by all transaction managers that coordinate virtual-compensable Web services. This interface defines abstract operations that a transaction manager service must support in order to participate in the 2PC coordination along a given mediator service.

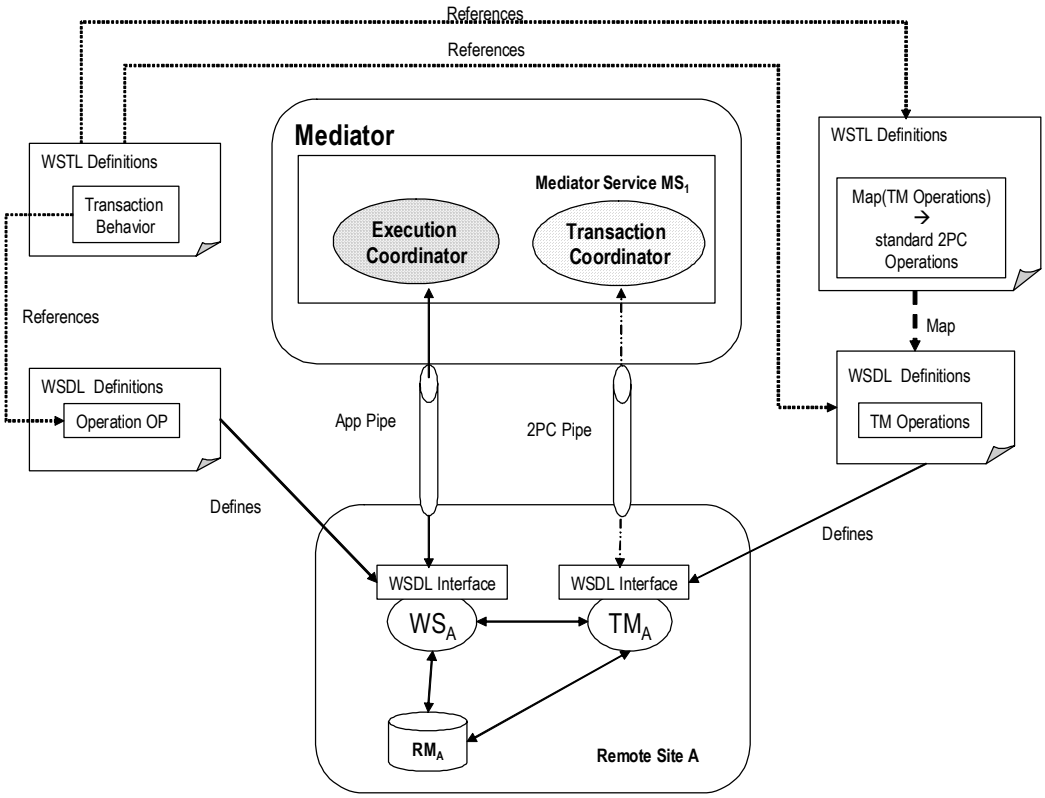


Figure 4.20 - WSTL framework for integrating virtual-compensable operations.

Figure 4.20 shows a Web service WS_A that has a virtual-compensable operation op, which is coordinated by a transaction manager TM_A. Both interfaces of Web service WS_A and transaction manager TM_A are described by WSDL documents. Besides its WSDL document, the Web service WS_A has a WSTL document that defines the transaction behavior of operation op. The transaction behavior of operation op references two documents. One is the WSDL document that defines the interface for accessing the transaction manager TM_A. The other is a WSTL document that maps the WSDL operations of TM_A in a standard 2PC coordinating interface. This is necessary because each transaction manager can have its own proprietary interface to coordinate the 2PC protocol, thus it is necessary to provide a mapping between the transaction manager proprietary operations and a set of standard operations known by the mediator service.

Using the information described above, whenever mediator service MS_1 invokes the operation op on behalf of a global transaction T, it knows that it must involve transaction manager TM_A in the coordination of T. It also knows that the communication with TM_A should be done following the WSDL definitions for TM_A. Finally, mediator service MS_1

will use the map information described in the WSTL map document for TM_A to infer which is the correct operation to invoke, at each step of the 2PC protocol.

We have described the general framework for integrating virtual-compensable operations. In the following sections, we will describe in detail how virtual-compensable operations are described by the WSTL framework.

4.2.9.1 Specifying Virtual-compensable Operations Through WSTL Elements

Recall from Section 4.2.5 that the `wstl:transactionbehavior` element supports two kinds of definitions (Figure 4.8): `Compensable`, and `VirtualCompensable` definitions. The `Compensable` definition was discussed in Section 4.2.6 through 4.2.7 and it describes the actions that should be taken to compensate a compensable operation. Similarly, the `VirtualCompensable` definition describes the actions that should be taken to compensate a *virtual-compensable operation*. As a virtual-compensable operation is not actually compensated but rather, their transaction support is exploited to provide the *semantics* of compensation, the `VirtualCompensable` definition indicates how WebTransact can exploit the local transaction support of a given virtual-compensable operation.

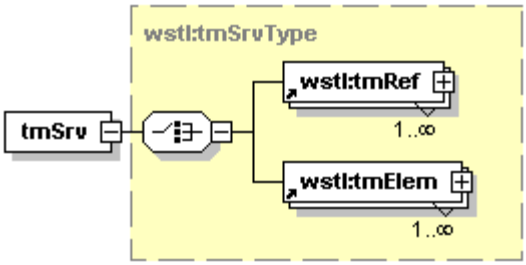


Figure 4.21 - The `tmSrv` element.

In WSTL framework, the `VirtualCompensable` definition is represented by the `tmSrv` element (Figure 4.21). This element can contain a non empty set of `tmRef` elements or a non empty set of `tmElem` elements as defined in the XML Schema fragment of Figure 4.22.

```
<xsd:element name="tmSrv" type="wstl:tmSrvType"/>

<xsd:complexType name="tmSrvType">
 <xsd:choice>
  <xsd:element ref="wstl:tmRef" maxOccurs="unbounded"/>
  <xsd:element ref="wstl:tmElem" maxOccurs="unbounded"/>
 </xsd:choice>
```

```
</xsd:complexType>
```

Figure 4.22 - XML schema fragment for `tmSrv` element, and `tmSrvType` type.

The `wstl:tmElem` element is of complex type `wstl:tmElemType` (Figure 4.23) having three attributes: `name`, `tmPort`, and `tmMap`. The optional attribute `name` provides a unique name among all `tmElemType` elements within the enclosing WSTL document. The mandatory `tmPort` attribute refers to a `wsdl:port` element using a `Qname`. The mandatory `tmMap` attribute refers to an `tmMap:tmSrvMap` element using a `Qname`. The `Qname` used by both attributes follows the linking rules defined by WSDL [137]. The value of the attribute `tmPort` references a port of a transaction manager service that can be accessed to communicate with a remote transaction manager that provides transaction support for the operation of the `transactionBehavior` element that contains the selected `wstl:tmElem`. The value of the attribute `tmMap` references the `tmMap:tmSrvMap` element of the WSTL framework. This element contains information on the mapping between the operation of the specific transaction manager interface and the standard operations known by WebTransact. The next section discusses this mapping in detail.

```
<xsd:element name="tmElem" type="wstl:tmElemType"/>
<xsd:complexType name="tmElemType">
 <xsd:complexContent>
  <xsd:extension base="wsdl:documented">
   <xsd:attribute name="name" type="xsd:NCName" use="optional"/>
   <xsd:attribute name="tmPort" type="xsd:QName" use="required"/>
   <xsd:attribute name="tmMap" type="xsd:QName" use="required"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Figure 4.23 - XML schema fragment for `tmElem` element, and `tmElemType` type.

4.2.9.2 Mapping WSDL operations of Remote Transaction Managers

Remote transaction managers are integrated in the WSTL framework through their 2PC WSDL interface. As the 2PC WSDL interface of remote transaction managers is defined according to local rules of each remote site, it is likely to exist different 2PC WSDL interfaces, probably one different interface for each remote transaction manager. In order to implement the communication between WebTransact and the remote transaction managers it is necessary to define rules for homogenizing these interfaces. To address this issue, the WSTL framework specifies a standard set of operations that must be supported by all remote transaction managers that should be integrated in

WebTransact. Each one of these standard operations is semantically related to a 2PC protocol command.

The WSTL framework does not specify a new 2PC protocol tailored for Web services environments. In fact, the Transaction Internet Protocol (TIP) (Section 4.2.9.3) ([35], [36]) is used to coordinate transaction synchronization messages between remote transaction managers and WebTransact. The TIP protocol is a new version of the well-known presumed abort 2PC protocol [92], which is specifically tailored for internet applications. The TIP protocol support the two-pipe model, thus satisfying the key requirement of transaction processing for Web services environments, i.e., the separation of the transaction synchronization messages from the application communication protocol. The WSTL framework makes use of most of the TIP specification, modifying only the way 2PC commands are formatted. In TIP, all commands and responses consist of one line of ASCII text, while in the WSTL framework the format of commands and responses is defined throughout WSDL operations and their respective bindings. Defining TIP commands at a higher degree of abstraction provides more flexibility when using the TIP protocol. Any transaction manager that supports the TIP protocol (or even any flavor of the presumed abort 2PC protocol) can define abstract operations through a WSDL interface, map these operations to the standard TIP commands, and then define how these abstract operations must be sent using WSDL binding definitions. Therefore, the TIP protocol plays the role of describing the semantics of the 2PC protocol along its abstract commands, while the concrete commands of the 2PC protocol are defined through WSDL definitions. In the next section, we review the main features of the TIP protocol.

4.2.9.3 The Transaction Internet Protocol

The goal of this section is to brief review the main features of the TIP protocol, the complete specification can be found in [35] and [36].

TIP is based on TCP/IP and supports the one-phase and the two-phase commit protocols with the presumed abort heuristics [92]. For every node involved in a transaction, a separate TCP/IP connection must be established. The main objective of TIP lays in the definition of state machines for the sites involved in a transaction in order to achieve a common state valid for the transactional connection.

TIP uses the two-pipe model, where transaction synchronization messages and the application messages are detached. The communication channel for synchronization between the transaction managers using TIP is session-oriented. It is possible to send different transactions over one TIP connection between two transaction managers. On the other hand, the applications can communicate with other protocols or paradigms, e.g. Java-RMI [126], DCOM [84], or CORBA [97].

TIP supports both the pushing and pulling models (Section 2.2.2) for propagating a transaction between two sites. In the pushing model, the transaction manager (TM) of the initiating site propagates a transaction identifier to a site that participates in the transaction, before the server component residing on that site become involved. In contrast to this approach, in the pulling model the client addresses a server component on a remote site, which then demands a new transaction identifier from its transaction manager. This transaction manager now pulls the identifier from the initiating site. In both cases, the transaction manager that opens the TCP/IP connection for TIP is called the primary site, and the responding transaction manager is the secondary site.

The communication in TIP is based on a command-response protocol defined as plain ASCII strings with valid ASCII characters from 32 to 127. The primary site sends a command and the secondary site selects a response related to the command. TIP defines commands either pertaining to a connection (IDENTIFY, MULTIPLEX, TLS) or to a transaction (ABORT, BEGIN, COMMIT, PREPARE, PULL, PUSH, QUERY, RECONNECT). These commands are discussed in Appendix 5.

The protocol defines several states in which a site can reside. For each state, a subset of protocol commands is allowed to perform a transition into another state. It is important to note that these are states of the sites regarding to the connection and not regarding to the transaction. The valid states of a TIP connection are:

- INITIAL: This is the first state of the TIP state machine after a connection is established between the transaction managers. In this state, the transaction managers consent on a protocol version.

- IDLE: The primary and the secondary have consent on a protocol version, and the primary has supplied an identifier to the secondary for reconnecting after a failure. There is no transaction associated with the connection in this state.

- ENLISTED: The connection is now associated with an active transaction, which can be completed by a one-phase or two-phase protocol.

- PREPARED: A connection is associated with a transaction that has been prepared.

- MULTIPLEXING: The connection is being used by a multiplexing protocol, which provides its own set of connections.

- ERROR: An error occurred and the transaction associated with the connection will be aborted. If the connection comes from PREPARED, a new connection must be established to complete the commit.

- BEGUN: The connection is associated with an active transaction, which can only be completed by a one-phase protocol.

All commands that are invalid in the current state result in a transition into the ERROR state. The error state can also be reached by sending an ERROR response command. Enhancements of TIP to the ordinary 2PC protocol are the possibility of reconnecting a previously disrupted connection and, in version 3.0 of TIP, it is possible to multiplex transactions over a single TCP/IP connection. Transactions are identified in TIP by transaction identifiers. TIP does not specify how a transaction identifier must be constructed. Actually, a transaction identifier can be any valid ASCII string. If a transaction identifier can be represented in or transformed to ASCII notation then it can be propagated to different transaction processing systems.

The WSTL framework uses the TIP protocol for synchronizing transaction messages when a mediator service needs to interact with virtual-compensable remote operations. Therefore, the remote transaction managers are supposed to be TIP-aware, i.e., they have a built-in TIP machine. The drawback of this solution is that the remote transaction manager itself must be aware of a specific TIP implementation. Fortunately, TIP can be used in conjunction with other distributed transaction models like the X/Open Distributed Transaction Processing (DTP) Model [130]. The X/Open DTP model defines four components: *i*) Application Program (AP); *ii*) Transaction Manager (TM); *iii*) Resource Manager (RM); and *iv*) Communication Resource Manager (CRM). In this model, TIP defines a TM-to-TM interoperability protocol, which is independent of the

application communication protocols. Application programming interfaces between the Application Programs (server component) and Transaction Managers/Resource Managers are unaffected by TIP. In the X/Open DTP model, the XA interface defines the TM to RM interaction. As TIP is compatible with the XA interface, a TIP transaction can involve application programs accessing multiple resource managers where the XA interface is being used to coordinate the transaction branches of resource managers.

It is important to note that most of today's distributed transaction systems ([6], [11], [62], [86], [117], [120]) have defined XA-compliant interfaces for interacting with XA resource managers. The compatibility between TIP and the XA interface of the X/Open DTP model offers the possibility of using TIP as the interoperability transaction protocol among a wide range of remote transaction managers. Even remote transaction managers that are not TIP aware, but are XA-compliant, can be easily integrated in WebTransact. All that is needed is a component that implements the TIP state machine and an additional API between the remote transaction manager and this TIP state machine (Figure 4.24).

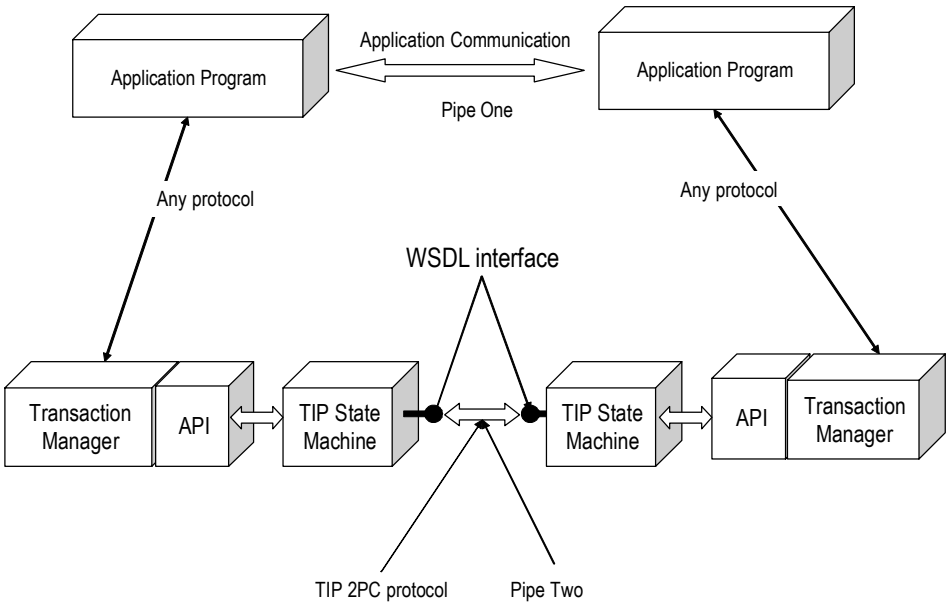


Figure 4.24 - Using TIP with Remote Transaction Managers.

In this section, we have presented a brief description of TIP. The WSTL framework uses TIP for coordinating transactions synchronizations messages exchanged between remote transaction managers and WebTransact. One way of doing this exchange of

messages is to define a standard format for these messages and then, enforce each remote transaction manager to support this standard format. Enforcing any kind of rigid format for message exchange is not realistic when considering a Web service environment. To provide more flexibility, the WSTL framework does not enforce a rigid format for communicating TIP commands. Instead, WSTL provides an XML Schema for mapping the TIP commands of a given transaction manager to the message format known by the WebTransact. Next Section describes this XML Schema for mapping TIP messages of remote transaction managers.

4.2.9.4 WSTL mapping for TIP Commands

As described in previous sections, the WSTL framework uses the TIP protocol for synchronizing transaction messages when a mediator service needs to interact with virtual-compensable remote operations. More precisely, TIP is used as an abstract protocol. When integrating a remote transaction manager in the WSTL framework, it is necessary to implement a TIP state machine and to provide a WSDL document for describing the expected format for TIP commands. The use of WSDL interface provides another level of independency when considering the interaction between the WebTransact and the remote TIP state machine. The only requirement for integrating a remote transaction manager in the WSTL framework is the existence of a TIP compliant state machine. If a TIP remote state machine is present, the remote site administrator is free to define any format for the expected TIP commands. To accommodate different WSDL interfaces for communicating with the remote TIP state machines, the WSTL framework defines the `tmMap` XML Schema. This XML Schema provides a mapping from the TIP commands of a specific remote state machine to the TIP commands known by WebTransact. This mapping is used to format TIP commands when a mediator service needs to coordinate a transaction that involves the remote site. Next, the WSTL document for mapping TIP commands is described.

The WSTL document for mapping TIP commands is a set of `tmmap:tmSrvMap`[6] elements. Each `tmmap:tmSrvMap` element is related to one or more remote transaction managers (more specifically, their TIP state machine) and has one required attribute, `name`. The attribute `name` provides a unique name among all `tmmap:tmSrvMap` elements

[6] The XML Schema for mapping TIP commands is defined using the prefix tmmap with namespace "http://schemas.xmlsoap.org/wsdl/tmmap/".

within the enclosing WSTL document.. Besides the attribute `name`, the `tmmap:tmSrvMap` element has nine child elements: `tmmap:abort`, `tmmap:commit`, `tmmap:identify`, `tmmap:multiplex`, `tmmap:prepare`, `tmmap:pull`, `tmmap:push`, `tmmap:query`, `tmmap:reconnect`, and `tmmap:tls`. Each one of these elements contains mapping information that defines how is the expected format for TIP commands of a given remote transaction manager. All these elements are extension of the complex type `tmmap:operationType` having two child elements: the `tmmap:inputMsg`, and the `tmmap:outputMsg`.

```
<xsd:complexType name="operationType">
 <xsd:complexContent>
  <xsd:extension base="wsdl:documented">
   <xsd:attribute name="targetOperation" type="xsd:QName" use="required"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Figure 4.25 - XML schema fragment for `operationType` type.

The type `tmmap:operationType` is an extension of the `wsdl:documentation` element (a container for human readable documentation) and has one attribute: `targetOperation`. The `targetOperation` is of type `xsd:QName` and links a standard TIP command to a particular `wsdl:operation`, following the linking rules of WSDL framework [137]. This `wsdl:operation` is an abstract operation defined in a WSDL document that describes the local format of TIP commands for a given remote transaction manager. Figure 4.25 shows the XML Schema fragment of `operationType` type.

```
<xsd:complexType name="msgLinkType">
 <xsd:attribute name="targetmsg" type="xsd:QName" use="required"/>
</xsd:complexType>
```

Figure 4.26 - XML schema fragment for `msgLinkType` type.

The `tmmap:inputMsg` and the `tmmap:outputMsg` are both extensions of the `tmmap:msgLinkType` type. Figure 4.26 shows the XML Schema fragment of `operationType` type. The `tmmap:mmsgLinkType` type has one attribute: `targetMsg` of type `xsd:QName`. This attribute is required and references a particular `wsdl:message`, following the linking rules of WSDL framework. Recall from Section 2.6 that a `wsdl:operation` element can contain: a `wsdl:input` element, a

wsdl:output element, and a wsdl:fault element. These elements refer to wsdl:message elements. Therefore, the wsdl:message referenced by the targetMsg attribute is an abstract message contained in the wsdl:operation, which is referenced by the attribute targetOperation of the tmmap:operationType type. When a targetMsg attribute belongs to a tmmap:inputMsg element, the wsdl:message belongs to a wsdl:input element of the wsdl:operation. When the targetMsg attribute belongs to a tmmap:outputMsg element, the wsdl:message belongs to a wsdl:output or a wsdl:fault element of the wsdl:operation. Figure 4.27 shows the relationship among the elements of a WSDL interface for describing TIP commands (of a given remote transaction manager) and the tmmap mapping elements.

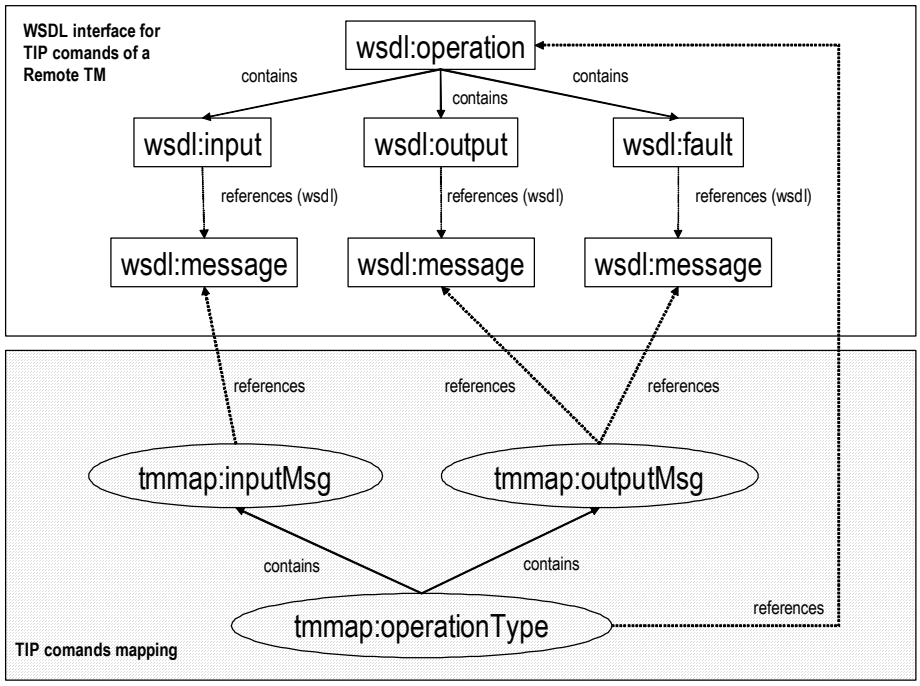


Figure 4.27- Relationship between tmmap elements and a WSDL interface for TIP commands.

In the previous paragraph, we have seen how the input, output, and fault messages of a WSDL operation are mapped in WSTL framework. What is still missing is how to map the parts that compose a WSDL message. This mapping is done using the tmmap:paramLinkType type. This type is a complex type that has two attributes: targetParam of type tmmap:XPathLinkType, and responseValue of type xsd:string. The targetParam attribute identifies a parameter inside the

`wsdl:message` referred by the `tmmap:outputMsg` or the `tmmap:inputMsg` elements. The mapping rules employed for the `targetParam` attribute are the same employed for the `wstl:msgLinkLinkType` described in Section 4.2.6.1. The `responseValue` attribute optionally identifies the semantic of response values returned by a TIP command thus, it is applied only to the `tmmap:outputMsg` element. The usage of these elements is exemplified in Section 4.2.10.

Figure 4.28 presents a fragment of the `tmmap` XML Schema showing the abort TIP command mapping. The abort TIP command receives the input parameter `connectionId` of type string returning one output parameter, which has two possible values: `ABORTED` or `ERROR`. The input parameter `connectionId` identifies the connection that must be aborted. The response `ABORTED` indicates the transaction associated with the selected connection has successfully aborted. The response `ERROR` indicates that the command was issued in the wrong state, or was malformed. The goal of the abort TIP command mapping is to identify the correct encoding format of the abort command for a particular remote transaction manager and how decode its results.

```
<xsd:element name="abort" type="tmmap:abortType"/>
<xsd:complexType name="abortType">
  <xsd:complexContent>
    <xsd:extension base="tmmap:operationType">
      <xsd:sequence>
          <xsd:element name="inputMsg">
            <xsd:complexType>
              <xsd:complexContent>
                <xsd:extension base="tmmap:msgLinkType">
                  <xsd:sequence>
                    <xsd:element name="connectionId" type="tmmap:paramLinkType"/>
                  </xsd:sequence>
                </xsd:extension>
              </xsd:complexContent>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="outputMsg">
            <xsd:complexType>
              <xsd:complexContent>
                <xsd:extension base="tmmap:msgLinkType">
                  <xsd:sequence>
                    <xsd:element name="aborted" type="tmmap:paramLinkType"/>
                    <xsd:element name="error"  type="tmmap:paramLinkType"/>
                  </xsd:sequence>
                </xsd:extension>
              </xsd:complexContent>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Figure 4.28 - WSTL mapping for the abort TIP command.

The abort TIP command mapping is represented by the XML element named `abort` of type `tmmap:abortType`. The `tmmap:abortType` type is an extension of the `tmmap:operationType` type. Therefore, it has one attribute named `targetOperation`. The value of this attribute should reference a particular `wsdl:operation` that represents the abort TIP command of a given remote transaction manager. The `tmmap:abortType` type has two child elements: the `inputMsg`, and the `outputMsg`. Both elements are extensions of the `tmmap:msgLinkType` type. Therefore, they have one attribute named `targetMsg` of type `xsd:QName`. For the `inputMsg` element, the value of this attribute should reference the input `wsdl:message` of the `wsdl:operation` referenced by `targetOperation` attribute of the `tmmap:abort` element. For the `outputMsg` element, the value of the `targetMsg` attribute should reference the output or fault `wsdl:message` of the `wsdl:operation` referenced by the `targetOperation` attribute of the `tmmap:abort` element. The `inputMsg` element has one child element named `connectionId` of type `tmmap:paramLinkType`. Therefore, the `connectionId` element has two attributes: `targetParam` of type `tmmap:XPathLinkType`, and `responseValue` of type `xsd:string`. The value of the `targetParam` attribute should contain an XPath expression, as defined in Section 4.2.6.2, that identifies a simple type, which represents the connection identifier input parameter. The `responseValue` attribute does not apply to the `inputMsg` element. The `outputMsg` element has two child elements: `aborted` and `error`, both of type `tmmap:paramLinkType`. Therefore, both elements have two attributes: `targetParam` of type `tmmap:XPathLinkType`, and `responseValue` of type `xsd:string`. For the `aborted` element, the value of the `targetParam` attribute should contain an XPath expression, as defined in Section 4.2.6.2, that identifies a simple type, which represents the output parameter where a value semantically equivalent to `ABORTED` should return. Now, the `responseValue` attribute is required. The value of this attribute will indicate which is the exact value that represents the standard result value `ABORTED`. For instance, a remote transaction manager can return the string `OK` indicating the success of the abort command. For the `error` element, the value of the `targetParam` attribute should contain an XPath expression, also defined as in Section 4.2.6.2, that identifies a simple type, which represents the output parameter (of a `wsdl:output` or `wsdl:fault` operation) where a value semantically equivalent to `ERROR` should return. In this case, the `responseValue` attribute is not required. As the abort command has only one possible

error message, even when the success or failure of the abort command is returned in the same output message, any value different of the `responseValue` attribute value, of the `tmmap:aborted` element, will be treated as an error message.

In this section we have described the mapping of TIP commands of a given remote transaction manager. Only the abort TIP command was explained. For a complete explanation of all TIP commands, see Appendix 1.

4.2.10 Example of Virtual-compensable Operations

Figure 4.29 shows the WSDL definition of a simple service for managing bank accounts, extended by the WSTL framework. The accounting service supports three operations: `balance` , `deposit`, and `withdraw`. The `wstl:transactionDefinitions` element defines these three operations as having virtual-compensable transaction behavior. The `wstl:transactionBehavior` element of each operation has a `wstl:tmRef` child element. This element references the `wstl:tmElem` element named `tmSoapMap`. The value `tmSrv:tmSrvSoapPort` of the `tmPort` attribute (of the `tmSoapMap` element) indicates the port to access the transaction manager that coordinate local transactions of the accounting service operations. While the value `tmSrvMap:tmSrvSoapmap` of the `tmMap` attribute (of the `tmSoapMap` element) indicates the WSTL document that contains the mapping information on the specific TIP commands employed by the transaction manager accessed through port `tmSrv:tmSrvSoapPort`. The prefix `tmSrv` indicates the namespace of the WSDL document that exposes the TIP commands of the remote transaction manager. The prefix `tmSrvMap` indicates the namespace for the WSTL document containing the TIP commands mapping information.

```
<definitions
  targetNamespace="http://localhost/bank/account/"
  ...
  xmlns:tmSrv="http://localhost/TMSrv/"
  xmlns:tmSrvMap="http://localhost/TMSrvMap/" >
  ...

  <portType name="AccountSoapPort">
    <operation name="balance">
      <input message="tns:balanceSoapIn"/>
      <output message="tns:balanceSoapOut"/>
    </operation>
    <operation name="deposit">
      <input message="tns:depositSoapIn"/>
      <output message="tns:depositSoapOut"/>
    </operation>
    <operation name="withdraw">
      <input message="tns:withdrawSoapIn"/>
```

```
      <output message="tns:withdrawSoapOut"/>
    </operation>
  </portType>

  <binding name="AccountSoapBinding" type="tns:AccountSoapPort">
  ...

  </binding>

  <service name="Accounting">
    <port name="AccountSoap" binding="tns:AccountSoapBinding">
      <soap:address location="http://localhost/bank/account.asmx"/>
    </port>
  </service>

  <wstl:trasactionDefinitions>
    <wstl:transactionBehavior type="virtualCompensable"
                              operationName="tns:balance">
      <wstl:tmSrv>  <wstl:tmRef tmElemName="tns:tmSoapMap"/> </wstl:tmSrv>
    </wstl:transactionBehavior>
    <wstl:transactionBehavior type="virtualCompensable"
                              operationName="tns:deposit">
      <wstl:tmSrv> <wstl:tmRef tmElemName="tns:tmSoapMap"/> </wstl:tmSrv>
    </wstl:transactionBehavior>
    <wstl:transactionBehavior type="virtualCompensable"
                              operationName="tns:withdraw">
      <wstl:tmSrv> <wstl:tmRef tmElemName="tmSoapMap"/> </wstl:tmSrv>
    </wstl:transactionBehavior>
    <wstl:tmElem name="tmSoapMap" tmPort="tmSrv:TMSrvSoapPort"
                 tmMap="tmSrvMap:TMSrvSoapMap"/>
  </wstl:trasactionDefinitions>
</definitions>
```

Figure 4.29 - The `Accounting` Service.

Figure 4.30 shows a fragment of the WSDL document that defines the `TMSrv` service, which represents the TIP interface of the remote transaction manager that coordinates the accounting service described above.

```
<definitions
 targetNamespace="http://localhost/TMSrv/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tmSrv="http://localhost/TMSvr/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
  <xsd:schema targetNamespace="http://localhost/TMSrv/">
   <xsd:element name="abort">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="pConnectionId" type="xsd:string" />
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
   <xsd:element name="abortResponse">
    <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="abortResult" type="tmSrv:resultType" />
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
   <xsd:complexType name="resultType">
    <xsd:sequence>
     <xsd:element name="resultSuccess" type="xsd:string" />
     <xsd:element name="resultError" type="xsd:string" />
    </xsd:sequence>
   </xsd:complexType>
   ...
  </types>
  <message name="abortSoapIn">
   <part name="parameters" element="tmSrv:abort" />
```

```
 </message>
 <message name="abortSoapOut">
  <part name="parameters" element="tmSrv:abortResponse" />
 </message>
 ...

 <portType name="TMSrvSoapPortType">
 <operation name="abort">
  <input message="tmSrv:abortSoapIn" />
  <output message="tmSrv:abortSoapOut" />
 </operation>
 ...

 </portType>
 ....

<binding name="TMSrvSoapBinding" type="tmSrv:TMSrvSoapPortType">
 <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                style="document" />
 <operation name="abort">
    ...
  </operation>
  ...

</binding>
<service name="TMSrv">
 <port name="TMSrvSoapPort" binding="tmSrv:TMSrvSoapBinding">
  <soap:address location="http://localhost/TMSrv/tmsrv.asmx" />
 </port>
</service>
</definitions>
```

Figure 4.30 -Example of a remote transaction manager interface expressed in WSDL. Only the abort TIP command is showed.

The `TMSrv` service provides only one service defined by the `wsdl:port` element `TMSrvSoapPort`, which supports the SOAP protocol (defined by the `wsdl:binding` element `TMSrvSoapBinding`). This port was referenced by the `wstl:transactionBehavior` elements of the accounting service operations. The `wsdl:portType` element `TMSrvSoapPortType` defines the abstract interface supported by the `TMSrv` service. This interface represents the format of the TIP commands expected by the remote transaction manager. In the example, only one TIP command is showed: the abort command, which is represented by the `abort wsdl:operation` of the `TMSrvSoapPortType` element. The `abort wsdl:operation` has the `abortSoapIn` and the `abortSoapOut` elements as its input and output messages respectively. The `wsdl:message abortSoapIn` element has one part that references the `tns:abort` element. The `tns:abort` element is defined as a complex type that has one child element, named `pConnectionId` of type `xsd:string`. The `wsdl:message abortSoapOut` element has one part that references the `tns:abortResponse` element. The `tns:abortResponse` element is defined as a complex type that has one child element, named `abortResult` of type `tns:resultType`. The `tns:resultType` is a

complex type that has two child elements: `resultSuccess` and `resultError`, both of type `xsd:string`. The definitions described above show how the abort TIP command must be formatted when interacting with a remote transaction manager that is accessed throughout the `TMSrv` service. Considering the WSDL definitions described above, Figure 4.31 and Figure 4.32 show the format of the messages expected by `TMSrv` service when invoking the abort TIP command.

```
POST /TMSrv/tmsrv.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://localhost/TMSrv/abort"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <abort xmlns="http://localhost/TMSrv/">
   <pConnectionId>1234</pConnectionId>
  </abort>
 </soap:Body>
</soap:Envelope>
```

Figure 4.31 - Expected SOAP input message for abort TIP command.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <abortResponse xmlns="http://localhost/TMSrv/">
   <abortResult>
    <resultSuccess>abortOk</resultSuccess>
    <resultError>null</resultError>
   </abortResult>
  </abortResponse>
 </soap:Body>
</soap:Envelope>
```

Figure 4.32 - Expected SOAP output message for the abort TIP command.

WebTransact uses the WSTL document of Figure 4.33 to generate the correct format of messages when invoking the abort command of the `TMSrv` service. This document contains one `tmMap:TMSrvMap` element named `TMSrvSoapMap`. The `TMSrvSoapMap` element was referenced by the `wstl:transactionBehavior` elements of the accounting service operations (Figure 4.29). The value of the `targetTMSrvPortType` attribute of the `TMSrvSoapMap` element indicates that the port type named `TMSrvSoapPortType` of the WSDL document with namespace

`http://localhost/TMSrv/` is being mapped. The value of the attribute `targetOperation` of `tmmap:abort` element indicates that the local identifier of TIP command abort is `abort`. The `tmmap:inputMsg` and `tmmap:outputMsg` elements indicate how to format the input message for invoking the abort TIP command and how to decode the returning values. For the `tmmap:inputMsg`, the value of the attribute `targetMsg` indicates that the local identifier of the input message for the TIP command abort is `abortSoapIn`. The value `abort/@pConnectionId` of the attribute `targetParm` of the `tmmap:connectionId` element indicates the XPath expression that locates the local identifier of the input parameter `connenctionId` of the TIP command abort. For the `tmmap:outputMsg`, the value of the attribute `targetMsg` indicates that the local identifier of the output message for the TIP command abort is `abortSoapOut`. The value `abortResponse/abortResult/@resultSuccess` of the attribute `targetParm` of the `tmmap:aborted` element indicates the XPath expression that locates the local output parameter for the aborted TIP response value. The value of the attribute `responseValue` indicates that the local response for a successful abort is `ABORT_OK`. Finally, the value `abortResponse/abortResult/@resultError` of the attribute `targetParm` of the `tmmap:error` element indicates the XPath expression that locates the local output parameter for returning error messages for the abort command.

```
<definitions
  xmlns:tmmap="http://schemas.xmlsoap.org/wsdl/tmmap/"
  xmlns:tmSrvMap="http://localhost/TMSrv/"
  xmlns:tmsrv="http://localhost/TMSrv/">

 <tmmap:TMSrvMap name="TMSrvSoapMap"
                    targetTMSrvPortType="tmsrv:TMSrvSoapPortType">
  <tmmap:abort targetOperation="abort">
    <tmmap:inputMsg targetMsg="abortSoapIn">
      <tmmap:connectionId targetParam="abort/@pConnectionId"/>
    </tmmap:inputMsg>
    <tmmap:outputMsg targetMsg="abortSoapOut">
      <tmmap:aborted targetParam="abortResponse/abortResult/@resultSuccess"
                     responseValue="ABORTED_OK"/>
      <tmmap:error targetParam="abortResponse/abortResult/@resultError"/>
    </tmmap:outputMsg>
  </tmmap:abort>
   ...

  </tmmap:TMSrvMap>
</definitions>
```

Figure 4.33 - Mapping example for the TIP abort command.

5. Mediator and Remote Services in WebTransact

In Chapter 4, we have shown how to describe different types of transaction behavior of Web services through WSTL elements. In this chapter, we explain how these Web services are integrated in the WebTransact model. The first step for integrating a Web service is to import its WSDL and WSTL definitions into the mediator repository. The next step is the definition of the remote service that links the imported Web service to a mediator service, semantically related to it. Remote services provide both content and mapping information. The content information defines the Web service capability to handle client requests. The mapping information provides a means by which messages are converted between the mediator service format and the Web service format. After the specification of its remote service, the Web service is ready to be used by WebTransact compositions.

In the next section, we describe the integration of Web services in the WebTransact model. First, we define how to describe a mediator service. Next, we define how to specify a remote service. In the last Section, we show an example that illustrates the concepts defined in this chapter.

5.1 MEDIATOR SERVICE DEFINITION

Mediator services are services used in specifying Web services compositions. Unlike remote services, which are logical units of work that perform remote operations at a particular site, mediator services are *virtual* services responsible for delegating its operations execution to one or more remote services. This delegation is done over a set of semantically equivalent remote services aggregated by the mediator service. A mediator service hides the inherent heterogeneity and distribution of remote services when specifying Web services compositions. To accomplish that, a mediator service exposes a single interface that is used by all compositions. This interface is constructed from all remote service interfaces aggregated by that mediator service.

The elements of the mediator service layer are described in the next sections using the Web Service Transaction Language (WSTL). Recall from Section 4.2 that WSTL is an XML language that supports the XML Schema specification (XSD) ([141], [142],

[143]). It is important to note that XML is being used here as a canonical model for specification purposes, only. The elements of the mediator service layer can be implemented using whichever model and repository one prefers. As these elements have a well-defined structure, they can be easily represented in different models and stored in different kind of repositories. If one chooses a relational database as the mediator repository, a natural choice for representing the elements of the mediator service layer would be the relational model [28].

The next paragraphs describe the WSTL element `mediatorService` that formally specifies a mediator service.

```
<xsd:element name="mediatorService" type="wstl:mediatorServiceType"/>
<xsd:complexType name="mediatorServiceType">
  <xsd:sequence>
    <xsd:element ref="wstl:operation" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:NCName" use="required"/>
</xsd:complexType>
```

Figure 5.1 - XML Schema fragment for the `mediatorService` element and the `mediatorServiceType` type.

The WSTL `mediatorService` element has one mandatory attribute named `id` (Figure 5.1). The attribute `id` provides a unique identifier among all `mediatorService` elements of the WebTransact repository. In addition to this attribute, the `mediatorService` element has a mandatory set of `operation` child elements (Figure 5.2). The set of `wstl:operation` elements defines the interface supported by the mediator service identified by `mediatorService/@id`.

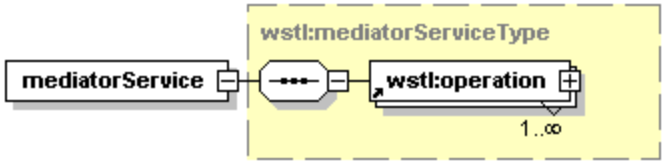


Figure 5.2 - WSTL `mediatorService` element.

Likewise the `wsdl:operation` element, the `wstl:operation` element can support four different primitives (Figure 5.3 and Figure 5.4):

- **One-way**. The mediator service operation receives an input message and does not send a response back.

- **Request-response**. The mediator service operation receives an input message and sends a response message back.

- **Solicit-response**. The mediator service operation sends an input message and receives a response back.

- **Notification**. The mediator service operation sends an input message and does not receive a response back.

```
<xsd:element name="operation" type="wstl:operationType"/>
<xsd:complexType name="operationType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:group ref="wstl:one-way"/>
      <xsd:group ref="wstl:request-response"/>
      <xsd:group ref="wstl:solicit-response"/>
      <xsd:group ref="wstl:notification"/>
    </xsd:choice>
    <xsd:element ref="wstl:contentDescription"
                 minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>
```

Figure 5.3 - XML Schema fragment for the `operation` element and the `operationType` type.

For all its supported primitives, the `wstl:operation` element has only one attribute named `name` of type `xsd:string`. The attribute `name` provides a unique name among all `operation` elements within a `wstl:mediatorService` element. In addition to this attribute, the `wstl:operation` element has a set of `contentDescription` child elements.

When the `wstl:operation` element is a one-way operation, it has only one child element named `inputMsg`. When the `wstl:operation` element represents a request-response operation, it has a sequence of three child elements: `inputMsg`, `outputMsg`, and `faultMsg`. When the `wstl:operation` element represents a solicit-response operation, it has a sequence of the same three child elements appearing in the request-response operation. The difference is in the order the elements appear: `outputMsg`, `inputMsg`, and `faultMsg`. Finally, when the `wstl:operation` element is a notification operation, it has only one child element named `outputMsg`. For all types of operations, the `faultMsg` element is optional. The `inputMsg` element defines the message format for the input parameters of a mediator service operation. The `outputMsg`

element defines the message format for the output parameters of a mediator service operation. The `faultMsg` element defines the format for the error messages raised by a mediator service operation.

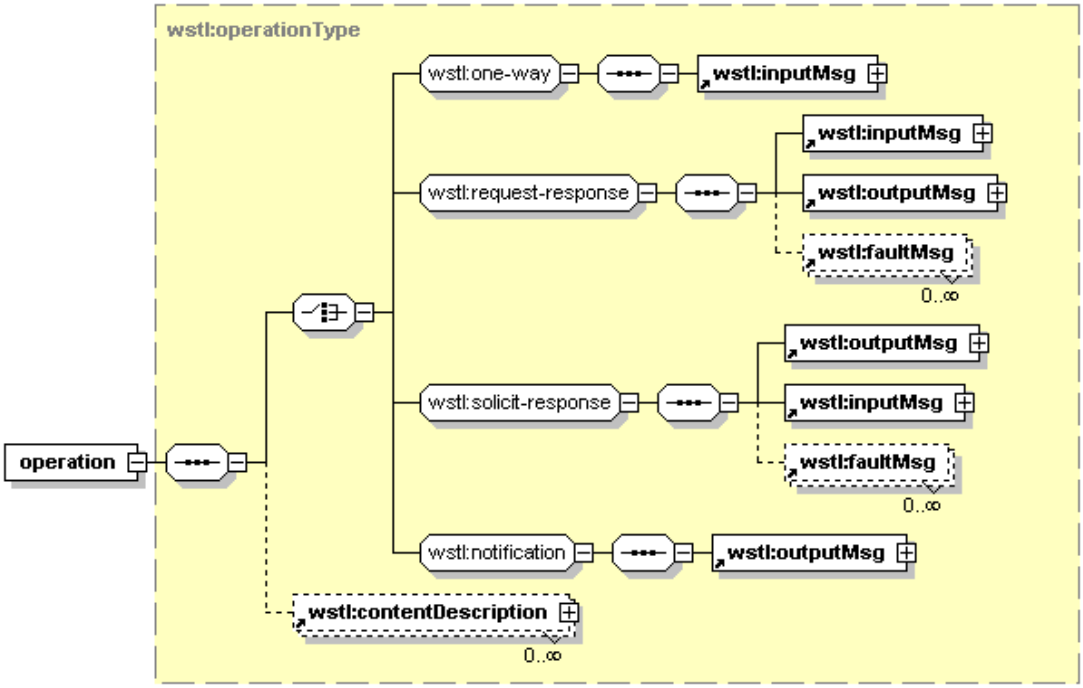


Figure 5.4 - WSTL `operation` element.

The `wstl:outputMsg` and `wstl:inputMsg` elements are of type `wstl:msgType` (Figure 5.5). The `wstl:msgType` type has only one optional attribute: `name` of type `xsd:NCName`. The attribute `name` provides a unique name among all `inputMsg` or `outputMsg` elements within a `wstl:mediatorService` element. In addition to this attribute, the `wstl:msgType` type has a set of `param` child elements. Each `wstl:param` element defines the format of a parameter of an input or output message of a mediator service operation.

```
<xsd:element name="inputMsg" type="wstl:msgType"/>
<xsd:element name="outputMsg" type="wstl:msgType"/>
<xsd:complexType name="msgType">
 <xsd:sequence>
  <xsd:element ref="wstl:param" minOccurs="0" maxOccurs="unbounded"/>
 </xsd:sequence>
 <xsd:attribute name="name" type="xsd:NCName" use="optional"/>
</xsd:complexType>
```

Figure 5.5 - XML Schema fragment for the `inputMsg` and `outputMsg` elements and the `msgType` type.

The wstl:param element has three mandatory attributes: name, type, and use (Figure 5.6). The attribute name provides a unique name among all param elements within a wstl:inputMsg or a wstl:outputMsg element. The attribute type refers to a XSD simple type or complex type using a QName. The attribute use indicates whether the parameter is required or optional when invoking the enclosing mediator service operation.

```
<xsd:element name="param" type="wstl:paramType"/>
<xsd:complexType name="paramType">
  <xsd:attribute name="name" type="xsd:NCName" use="required"/>
  <xsd:attribute name="type" type="xsd:QName" use="required"/>
  <xsd:attribute name="use" use="optional">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="required"/>
        <xsd:enumeration value="optional"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>
```

Figure 5.6 - XML Schema fragment for the param element and the paramType type.

The wstl:contentDescription element defines the content description of a parameter of the enclosing mediator service operation. This element has one mandatory attribute named medParam. This attribute refers to a parameter belonging to the mediator service operation defined by the value of the attribute wstl:mediatorService/operation/@name. In addition to the medParam attribute, the contentDescription element has a set of domain child elements. The domain element has one attribute named value, of type xsd:string. The value of this attribute is an element of the domain of the attribute defined by medParam.

The content description constrains the possible values that can be bound to a parameter of a given mediator service operation, when invoking such operation. WebTransact makes use of this information to infer whether a mediator service operation is able to attend a particular application invocation. This prevents an application invocation to be forwarded to the mediator service layer when it is out of the mediator service range. The content description of all message parameters of a mediator service operation is a superset of the content description defined for all remote service operations aggregated by it (Section 5.2).

```
<xsd:element name="contentDescription">
  <xsd:complexType>
    <xsd:sequence maxOccurs="unbounded">
      <xsd:element ref="wstl:domain"/>
```

```
    </xsd:sequence>
    <xsd:attribute name="medParam" type="tmmap:XPathLinkType" use="required"/>
  </xsd:complexType>
</xsd:element>
<xsd:element name="domain">
  <xsd:complexType>
    <xsd:attribute name="value" type="xsd:string" use="required"/>
  </xsd:complexType>
</xsd:element>
```

Figure 5.7 - XML Schema fragment for the `contentDescription` and `domain` elements.

In the next section, we present the WSTL element that describes remote services. After that, we present an example to illustrate the definition of both remote services and mediator services using WSTL.

5.2 REMOTE SERVICE INTEGRATION

WSDL definitions provide information on the abstract and concrete format of operations supported by a given remote Web service, as well as the internet address for accessing them. In WebTransact, each WSDL port type - and not a WSDL service - is imported as a new remote service. This decision was made based on mapping issues. As described in Chapter 3, WebTransact has a multilayered architecture of specialized components. Composition specifications are built upon mediator services. Mediator services are defined over distributed, heterogeneous, and autonomous remote services. The goal of the mediator service layer is to hide the inherent heterogeneity and distribution of remote services from the application level. A mediator service exposes a single interface that is used by composition specifications. As mediator services aggregate semantically equivalent remote services, which possibly have different interfaces, it is necessary to provide mapping information between the interface supported by the mediator service and each one of the interfaces supported by its aggregated remote services. Since the WSDL port type element defines the syntax for calling a set of remote operations, i.e., a specific supported interface, each WSDL port type definition is considered as a separated remote service.

The WSTL element `remoteService` defines remote services. This element links a mediator service to a WSDL port type element, provides mapping information between mediator service operations and port type operations, and specifies the content description of the remote service. The linking information defines the mediator service that aggregates the remote service. The mapping information prescribes how the input

parameters of a remote service operation are constructed from the input parameters of its related mediator service operation, as well as how the output parameters or fault messages received from that remote service operation are mapped to the output or fault messages of its related mediator service operation. The content description specifies whether a remote service is able to execute a particular service invocation.

For example, consider remote services rm_1 and rm_2 providing car reservations. Remote service rm_1 can make car reservations world wide, while remote service rm_2 accepts only car reservations in Brazil. Now, consider that mediator service ms_1 aggregates rm_1 and rm_2. If ms_1 receives a request to make a car reservation inside USA then, ms_1 will invoke only the remote service rm_1. The mediator service ms_1 knows, by using the remote service content description, that rm_2 is not able to make car reservation outside Brazil.

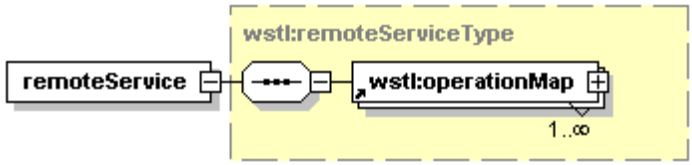


Figure 5.8 - WSTL `remoteService` element.

The WSTL `remoteService` element has three mandatory attributes: `id`, `medServ`, and `portType` (Figure 5.9). The attribute `id` provides a unique identifier among all `remoteService` elements of the WebTransact repository. The attribute `medServ` refers to a `wstl:mediatorService` element using a `Qname`. The attribute `portType` refers to a `wsdl:portType` element using a `Qname`. The `Qname` used by both attributes follows the linking rules defined by WSDL [137]. The value of the attribute `medServ` defines the mediator service that aggregates the remote service. The value of the attribute `portType` defines the abstract interface of the remote service. In addition to these attributes, the `remoteService` element has a mandatory set of `operationMap` child elements (Figure 5.8).

```
<xsd:element name="remoteService" type="wstl:remoteServiceType"/>
<xsd:complexType name="remoteServiceType">
  <xsd:sequence>
    <xsd:element ref="wstl:operationMap" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string" use="required"/>
  <xsd:attribute name="medServ" type="xsd:QName" use="required"/>
  <xsd:attribute name="porType" type="xsd:QName" use="required"/>
```

```
</xsd:complexType>
```

Figure 5.9 - XML Schema fragment for `remoteService` element, and `remoteServiceType` type.

Each `operationMap` element specifies mapping information between an operation of a port type element and its related mediator service operation as well as the content description of the operation of the port type element being mapped.

The `operationMap` element has three mandatory attributes: `name`, `ptOperation`, and `medOperation` (Figure 5.10). The attribute `name` provides a unique name among all `operationMap` elements within a `wstl:remoteService` element. The attribute `ptOperation` refers to a `wsdl:operation` using a `Qname`. This `wsdl:operation` belongs to the port type element defined by the value of the attribute `wstl:remoteService/@portType`[7]. The attribute `medOperation` refers to a `wstl:operation` using a `Qname`. This `wstl:operation` belongs to the mediator service element defined by the value of the attribute `wstl:remoteService/@medServ`. The `Qname` used by both attributes follows the linking rules defined by WSDL [137]. The values of the attributes `ptOperation` and `medOperation` define a relation R = (`ptOperation`, `medOperation`) indicating that the remote service operation `ptOperation` implements the mediator service operation `medOperation`.

```
<xsd:element name="operationMap" type="wstl:operationMapType"/>
<xsd:complexType name="operationMapType">
  <xsd:sequence>
    <xsd:element ref="wstl:inputMap" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="wstl:outPutMap" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="wstl:faultMap" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="wstl:contentDescription" minOccurs="0"
                                                  maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:NCName" use="required"/>
  <xsd:attribute name="ptOperation" type="xsd:QName" use="required"/>
  <xsd:attribute name="medOperation" type="xsd:QName" use="required"/>
</xsd:complexType>
```

Figure 5.10 - XML Schema fragment for `operationMap` element, and `operationMapType` type.

In addition to the attributes described above, the `wstl:operationMap` element has three optional child elements: `inputMap`, `outputMap`, `faultMap`, and

[7] We are using the XPath notation as described in Section 4.2.6.1.

contentDescription (Figure 5.11). The first three elements specify mapping information. The fourth element specifies the content description of the operation being described. The inputMap , outputMap, and faultMap elements are of the same type wstl:messageMapType (Figure 5.13).

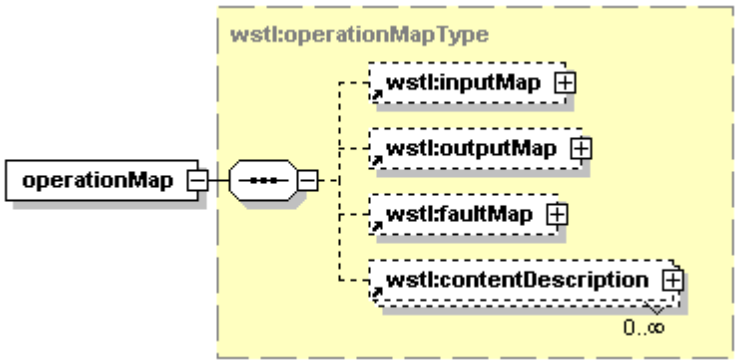


Figure 5.11 - WSTL operationMap element.

5.2.1 The inputMap element

The inputMap element prescribes how the fields in the input message part of the selected port type operation are constructed from the input parameters of its related mediator service operation. This element has a set of paramMap child elements. Each paramMap element specifies a parameter-mapping between the input parameters of the mediator service operation - defined by the value of the attribute wstl:remoteService/operationMap/@medOperation - and a particular field in the input message part of a port type operation - defined by the value of the attribute wstl:remoteService/operationMap/@ptOperation.

The wstl:paramMap element has two mandatory attributes: targetParam and mapFunction (Figure 5.13). The attribute targetParam refers to a particular field in the input message part of the operation defined by the value of the attribute wstl:remoteService/operationMap/@ptOperation. The value of this attribute is an XPath expression as described in Section 4.2.6.1. The attribute mapFunction references a function stored in the mediator repository. Before describing this attribute, let us describe the wstl:sourceParam element.

The sourceParam element is a child element of the wstl:paramMap element. Each sourceParam element has three optional attributes: XPath, fixed, and responseValue. When enclosed within a wstl:inputMap element, the XPath attribute refers to an input parameter of the mediator service operation defined by the value of the attribute wstl:remoteService/operationMap/@medOperation. The value of this attribute is an XPath expression as described in Section 4.2.6.1. The attributes fixed and responseValue do not apply when a sourceParam element is enclosed within an inputMap element.

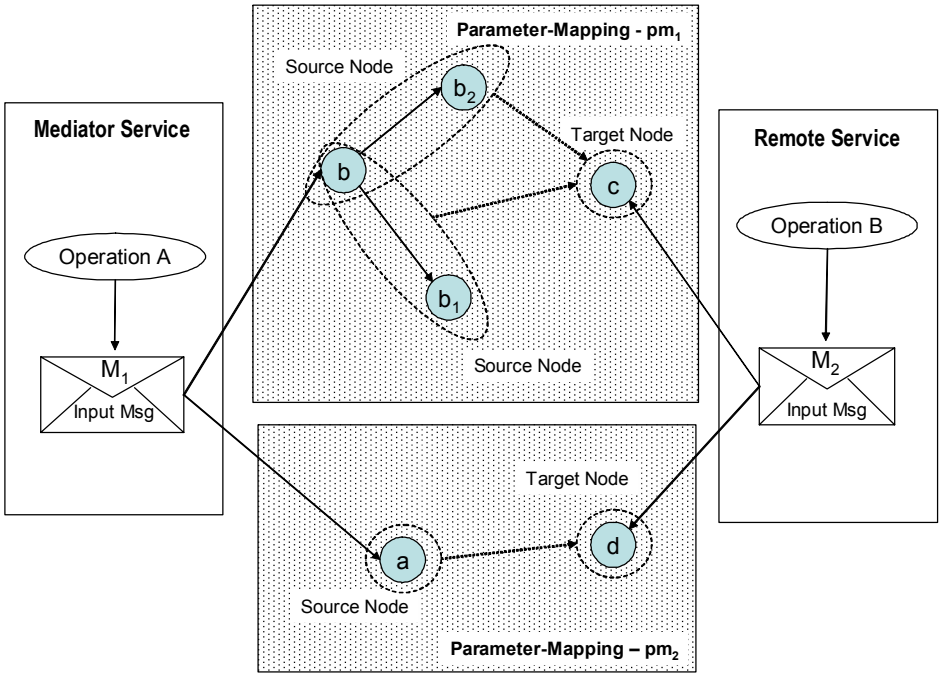


Figure 5.12 - Parameter-mapping example.

The parameter-mapping defined by a wstl:paramMap element can be represented by a graph with a target node, identified by the attribute targetParam of the wstl:paramMap element, and source nodes identified by their sourceParam child elements. The directed edges in the graph indicate the flow of data. For example, Figure 5.12 shows two parameter-mappings pm_1 and pm_2, between the input message parts of port type operation B and the input parameters of mediator service operation A. The parameter-mapping pm_1 is represented by a direct graph with one target node c and two source nodes: b_1 and b_2. The edges of this graph link the source nodes b_1 and b_2 to the target node c. This parameter-mapping indicates that the target node c is constructed from

source nodes b_1 and b_2. The parameter-mapping pm_2 is represented by a direct graph with one target node and one source node d. The edge of this graph links the source node a to the target node d. This parameter-mapping indicates that the target node a is constructed from source node d. The parameter-mappings pm_1 and pm_2 specify how the fields in the input message part of the remote service operation B are constructed from the input parameters of the mediator service operation A.

The rules for converting a set of input parameters of a mediator service operation into a field of an input message part of a port type operation are defined by the mapping function referenced in the mapFunction attribute. This mapping function is used to solve type and domain mismatches between the source nodes and the target node of a parameter-mapping. The mapping function is defined as $func : A \rightarrow B$, where: A is the set of values of the source nodes of the parameter-mapping; and B is a value belonging to the domain of the target node of the parameter-mapping. The graph that represents a parameter link can have zero, one, or many source nodes. If it has zero source nodes, no mapping is needed, thus there is no mapping function (*func* is null). Considering parameter-mapping enclosed in inputMap elements, this case represents port type operations that have input message parts with empty messages. Therefore, the related mediator service operation may have none input parameters. If the graph has only one source node, *func* will perform a one-to-one mapping between the source node and the target node. If the graph has more than one source node, *func* will perform a many-to-one mapping between the set of source nodes and the target node.

```
<xsd:element name="inputMap" type="wstl:paramMapType"/>
<xsd:element name="outPutMap" type="wstl: paramMapType"/>
<xsd:element name="faultMap" type="wstl: paramMapType"/>
<xsd:complexType name="msgMapType">
 <xsd:sequence>
  <xsd:element ref="wstl:paramMap" maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>
<xsd:element name="paramMap" type="wstl:paramMapType"/>
<xsd:complexType name=" paramMapType">
  <xsd:sequence>
    <xsd:element ref="wstl:sourceParam" minOccurs="unbounded" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="targetParam" type="tmmap:XPathLinkType" use="required"/>
  <xsd:attribute name="mapFunction" type="xsd:NCName" use="required"/>
</xsd:complexType>
<xsd:element name="sourceParam">
  <xsd:complexType>
    <xsd:attribute name="XPath" type="tmmap:XPathLinkType" use="required"/>
  </xsd:complexType>
</xsd:element>
```

Figure 5.13 - XML Schema fragment for elements: inputMap , outputMap, faultMap, and faultMap; and types: msgMapType and paramMapType.

5.2.2 The outputMap element

The `outputMap` element prescribes how the output parameters of a mediator service operation are constructed from the fields in the output message part of its related port type operation. Likewise the `wstl:inputMap` element, the `wstl:outputMap` element is of type `wstl:paramMapType` (Figure 5.13). Therefore, both elements have the same structure although different semantics.

For the `outputMap` element, each `paramMap` element specifies a parameter-mapping between a particular output parameter of a mediator service operation - defined by the value of the attribute `wstl:remoteService/operationMap/@medOperation` - and a set of fields in the output message part of a port type operation - defined by the value of the attribute `wstl:remoteService/operationMap/@ptOperation`. In the `wstl:outputMap` element, the attribute `targetParam` refers to a particular output parameter of the mediator service operation defined by the value of the attribute `wstl:remoteService/operationMap/@medOperation`. The attribute `mapFunction` also references a function stored in the mediator repository. The `XPath` attribute of the `sourceParam` element refers to a field in the output message part of the port type operation defined by the value of the attribute `wstl:remoteService/operationMap/@ptOperation`. For the `wstl:outputMap` element, the attribute `targetParam` along with the `sourceParam` elements also define a parameter-mapping. Now, the parameter-mapping defines a flow of data in the opposite direction, regarding the parameter-mapping defined by the `wstl:inputMap` element. In the `wstl:outputMap` element, the parameter-mapping is defined between a particular output parameter of a mediator service operation - defined by the value of the attribute `targetParam` - and a set of fields of an output message part - defined by the content of the `sourceParam` elements.

The rules for converting a set of fields of an output message part of a given port type operation into an output parameter of a mediator service operation are also defined by the mapping function referenced in the `mapFunction` attribute. For the `wstl:outputMap` element, the mapping function is defined as $func : A \rightarrow B$, where: A is the set of values of the fields in the output message part defined by the content of the `sourceParam` elements; and B is a value belonging to the domain of `targetParam`. One output parameter of a mediator service operation can be linked to one or more fields

in the output message part of a port type operation. If it is linked to only one field, *func* will perform a one-to-one mapping between the corresponding `targetParam` and `sourceParam` values. If it is linked to more than one field, *func* will perform a many-to-one mapping between the set of `sourceParam` and the corresponding `targetParam` values. Note that it is not possible to have an output parameter of a mediator service operation without a corresponding field in the output message part of its related port type operation. Mediator services are a superset of its aggregated remote services. Therefore, all information returned by a remote service must be mapped to the mediator service. Unlike the `wstl:inputMap` element, the `wstl:outputMap` element may enclose `sourceParam` elements containing the attribute `fixed` in place of the `XPath` attribute. When the attribute `fixed` appears, it indicates that there is no data link between the target output parameter of the selected mediator service operation and any fields of output message part of the selected port type element, i.e., there is no source fields. In this case, instead of constructing the target parameter from a set of values returned from the remote service, it must always be constructed using the content of the attribute `fixed`.

5.2.3 The faultMap element

The `faultMap` element matches the fault messages of a mediator service operation with the error messages raised by its related port type operation. An error message of a port type operation can be embedded in the fields of its fault *or* output message part. Therefore, the `faultMap` element must prescribe how the fault messages of a mediator service operation are constructed from the fields in the fault or output message part of its related port type operation. Likewise the `inputMap` and `outputMap` elements, the `wstl:faultputMap` element is of type `wstl:paramMapType` (Figure 5.13).

For the `faultMap` element, each `paramMap` element specifies a parameter-mapping between a particular fault message of the mediator service operation - defined by the value of the attribute `wstl:remoteService/operationMap/@medOperation` - and a field in the fault/output message part of a port type operation - defined by the value of the attribute `wstl:remoteService/operationMap/@ptOperation`. The attribute `targetParam` refers to a particular fault message of the selected mediator service operation. The attribute `mapFunction` does not apply when the `paramMap` element is enclosed in a `faultMap` element. The `XPath` attribute of the `sourceParam`

element refers to a field in the output or fault message part of the selected port type operation.

For the `wstl:faultMap` element, the attribute `targetParam` along with the `sourceParam` elements also define a parameter-mapping. However, in this case the parameter-mapping indicates fault messages. In the `wstl:faultMap` element, the parameter-mapping is defined between a particular fault message of a mediator service operation and a field of an output or fault message part of its related port type operation. Unlike the `wstl:inputMap` and `wstl:outputMap` elements, the `wstl:faultMap` element may enclose `sourceParam` elements containing the attribute `responseValue` along with the `XPath` attribute The `responseValue` attribute identifies the semantic of response values returned by an output or fault message of a port type operation. The usage of these elements is exemplified in Section 5.3.

5.2.4 The contentDescription element

Finally, the definition of the last child element of the `wstl:operationMap` element: the `contentDescription` element (see Section 5.1 and Figure 5.7). This element is the same element used for describing the content description of parameters of a mediator service operation (Section 5.1). However, when the `contentDescription` element is enclosed by a `wstl:operationMap` element, it describes the content of a remote service operation. The content description of a remote service operation is described concerning the domain of the input parameters of its related mediator service operation. Recall from Section 5.1 that the `contentDescription` element has one mandatory attribute named `medParam`. In this context, this attribute refers to an input parameter belonging to the mediator service operation defined by the value of the attribute `wstl:remoteService/operationMap/@medOperation`. In this case, the set of `domain` elements of a `contentDescription` element is defining a subset of the domain of the input parameter of the mediator service operation defined by the value of the attribute `wstl:remoteService/operationMap/@medOperation`.

In Sections 5.1 and 5.2, we have described how mediator services are specified as well as how remote Web services are integrated and mapped in the WSTL framework. The next section illustrates these concepts.

5.3 REMOTE SERVICE INTEGRATION EXAMPLE

In this section, we illustrate the definitions of a mediator service and the integration of a remote service using the WSTL framework.

The following example (Figure 5.14) shows a WSTL definition of a mediator service providing car reservations. The value `msCarReservation` of the attribute `mediatorService/@id` identifies the mediator service. This mediator service supports two operations: `carReserv` and `cancelCarReserv`. The `carReserv` operation makes a car reservation while the `cancelCarReserv` operation cancels a previous executed car reservation. In this section, we will show only the `carReserv` operation. The complete example, with the `cancelCarReserv` operation, is in Appendix 7.

```
<definitions ...

<types ...
  <schema ...
    <complexType name="locationType">
      <attribute name="country" type="string"/>
      <attribute name="state" type="string"/>
      <attribute name="city" type="string"/>
      <attribute name="office" type="string"/>
    </complexType>

    <complexType name="companiesType">
      <sequence>
        <element name="companyName" type="string" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </schema>
</types>

<wstl:mediatorService id="msCarReservation">
  <wstl:operation name="carReserv">
    <wstl:inputMsg>
      <wstl:param name="preferredCompany" type="lxsd:companiesType" use="optional"/>
      <wstl:param name="pickupDate" type="xsd:date" use="required"/>
      <wstl:param name="pickupLocation" type="lxsd:locationType" use="required"/>
      <wstl:param name="pickupTime" type="xsd:time" use="required"/>
      <wstl:param name="dropOffDate" type="xsd:date" use="required"/>
      <wstl:param name="dropOffLocation" type="lxsd:locationType" use="required"/>
      <wstl:param name="dropOffTime" type="xsd:time" use="required"/>
    </wstl:inputMsg>
    <wstl:outputMsg>
      <wstl:param name="reservationCode" type="xsd:string"/>
      <wstl:param name="company" type="xsd:string"/>
    </wstl:outputMsg>
    <wstl:faultMsg errorCode="ERROR_100" description="No available cars"/>
    <wstl:faultMsg errorCode="ERROR_101"
        description="No available remote services matching this invocation"/>
    <wstl:faultMsg errorCode="ERROR_102" description="Communication failure"/>
    <wstl:contentDescription medParam="inputMsg/preferredCompany/@companyName">
      <wstl:domain value="Avis"/>
      <wstl:domain value="Hertz"/>
      <wstl:domain value="Localiza"/>
      <wstl:domain value="BrazilCar"/>
    </wstl:contentDescription>
    <wstl:contentDescription medParam="inputMsg/pickUpLocation/@country">
      <wstl:domain value="USA"/>
      <wstl:domain value="Brazil"/>
      <wstl:domain value="Netherland"/>
      <wstl:domain value="Portugal"/>
    </wstl:contentDescription>
  </wstl:operation>
```

```
<wstl:operation name="cancelCarReserv" ...
</wstl:mediatorService>
</definitions>
```

Figure 5.14 - WSTL Schema fragment of the mediator service `msCarReservation`.

The `carReserv` operation has seven input parameters: `preferredCompany`, `pickupDate`, `pickupLocation`, `pickupTime`, `dropoffDate`, `dropoffLocation`, and `dropoffTime`. The parameter `preferredCompany` is of complex type `companyType`, which is a list of strings that stores company names. The parameters `pickupLocation`, and `dropoffLocation` are of complex type `LocationType`, which has four attributes `country`, `state`, `city`, and `office`, all of type `xsd:string`. The other parameters are of XSD simple types. The return values of the `carReserv` operation are wrapped in two output parameters: `reservationCode`, and `company`, both of type `xsd:string`. Besides the return values just described, the `carReserv` operation may return three fault messages identified by the value of the attribute `faultMsg/@errorCode`. The meaning of the input and output parameters are easily inferred from their names. The input parameters `preferredCompany` and `pickupLocation` of the `carReserv` operation have their content description defined. For the `preferredCompany` parameter, the content description specifies that this parameter can accept only `companyName` elements within the domain of values {`Avis`, `Herts`, `Localiza`, `BrazilCar`}. For the `pickupLocation` parameter, the content description specifies that the attribute `country`, of this parameter, can have only values within the domain of values {`USA`, `Brazil`, `Netherlands`, `Portugal`}. Therefore, the following requests will be reject by this mediator service:

- `preferredCompany/@companyName=″Alamo″`;
  `pickupLocation/@country=″Any″`

- `preferredCompany/@companyName=″Hertz″`;
  `pickupLocation/@country=″Argentina″`

- `preferredCompany/@companyName=″Any″`;
  `pickupLocation/@country=″Argentina″`

Now, consider the integration of the remote service `carReservation` described in Section 4.2.2. The WSDL Schema fragment in Figure 5.15 shows the port type, messages, and types definitions of this remote service.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...
  <types>
    ...
      <xsd:element name="reservation">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="pInput" type="tns:reservationInputType"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:complexType name="reservationInputType">
        <xsd:sequence>
          <xsd:element name="pickupInfo" type=”tns:infoType”/>
          <xsd:element name="dropoffInfo" type=”tns:infoType”/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="infoType">
        <xsd:sequence>
          <xsd:element name="dt" type="xsd:dateTime"/>
          <xsd:element name="location" type="xsd:string"/>
          <xsd:element name="time" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:element name="reservationResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="reservationResult" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="cancelReservation" ...

    </xsd:schema>
  </types>

  <message name="reservationSoapIn">
    <part name="parameters" element="lxsd:reservation"/>
  </message>
  <message name="reservationSoapOut">
    <part name="parameters" element="lxsd:reservationResponse"/>
  </message>
  <message name="cancelReservationSoapIn ...
  <portType name="reservationSoap">
    <operation name="reservation">
      <input message="tns:reservationSoapIn"/>
      <output message="tns:reservationSoapOut"/>
    </operation>
    <operation name="cancelReservation" ...
    </operation>
  </portType>
  ...

</definitions>
```

Figure 5.15 - WSDL Schema fragment of the car reservation remote service.

The WSDL definitions of the `carReservation` service specify only one port type element named `reservationSoap`. Therefore, the integration of this service in WebTransact will generate only one remote service, related to that port type element. Figure 5.16 shows the WSTL definition of the remote service that integrates the

carReservation service. The value rsBrazilCar of the attribute remoteService/@id identifies the remote service. The qualified value tns:msCarReservation of the attribute remoteService/@medServ links this remote service to a mediator service already stored in the WebTransact repository. Note that the value msCarReservation is the identifier of the mediator service formerly described (Figure 5.14). Therefore, the mediator service msCarReservation aggregates the remote service rsBrazilCar, which is being created. The qualified value rs:reservationSoap of the attribute remoteService/@portType links the WSTL remote service rsBrazilCar to the port type element, reservationSoap, of the WSDL remote service carReservation (Figure 5.15). The WSTL remote service rsBrazilCar must provide mapping information for all operations of the reservationSoap port type element, which are aggregated by the operations of the mediator service msCarReservation. In this example, all operations of that port type element are mapped to their related operations in the mediator service msCarReservation. We show only the mapping of the carReserv operation. The complete example, with the mapping of the cancelCarReserv operation, can be found in Appendix 7.

```
<wstl:remoteService id="rsBrazilCar"
                      medServ="tns:msCarReservation"
                      portType="rs:reservationSoap">
  <wstl:operationMap name="reservMap"
                      ptOperation="rs:reservation"
                      medOperation="tns:carReserv">
    <wstl:inputMap>
      <wstl:paramMap
             targetParam="reservationSoapIn/reservation/pInput/pickupInfo/@dt">
        <wstl:sourceParam XPath="@pickupDate"/>
      </wstl:paramMap>
      <wstl:paramMap
        targetParam="reservationSoapIn/reservation/pInput/pickupInfo/@location"
        mapFunction="concatValues">
          <wstl:sourceParam XPath="pickupLocation/@state"/>
          <wstl:sourceParam XPath="pickupLocation/@city"/>
          <wstl:sourceParam XPath="pickupLocation/@office"/>
      </wstl:paramMap>
      <wstl:paramMap
          targetParam="reservationSoapIn/reservation/pInput/pickupInfo/@time">
          <wstl:sourceParam XPath="@pickupTime"/>
      </wstl:paramMap>
      <wstl:paramMap
          targetParam="reservationSoapIn/reservation/pInput/dropOffInfo/@time">
          ...

    </wstl:inputMap>
    <wstl:outputMap>
      <wstl:paramMap targetParam="@reservationCode">
        <wstl:sourceParam
           XPath="reservationSoapOut/reservationResponse/@reservationResult"/>
      </wstl:paramMap>
      <wstl:paramMap targetParam="@company">
        <wstl:sourceParam
             fixed="BrazilCar"/>
      </wstl:paramMap>
    </wstl:outputMap>
```

```
    <wstl:faultMap>
      <wstl:paramMap targetParam="@ERROR_100">
        <wstl:sourceParam
            XPath="reservationSoapOut/reservationResponse/@reservationResult"
            responseValue="NO_AV_CARS"/>
      </wstl:paramMap>
    </wstl:faultMap>
    <wstl:contentDescription medParam="preferred/@company">
      <wstl:domain value="BrazilCar"/>
    </wstl:contentDescription>
    <wstl:contentDescription medParam="pickupLocation/@country">
      <wstl:domain value="Brazil"/>
    </wstl:contentDescription>
  </wstl:operationMap>
  <wstl:operationMap name="cancelReservMap" ...
  </wstl:operationMap>
</wstl:remoteService>
```

Figure 5.16 - WSTL Schema fragment of the remote service `rsBrazilCar`.

The `operationMap` element named `reservMap` defines a set of parameter-mappings between the message parts of the port type operation, `reservation` - defined by the value of the attribute `ptOperation` - and the parameters of the mediator service operation, `carReserv` - defined by the value of the attribute `medOperation`.

The `inputMap` element of the `reservMap` element prescribes how the fields in the input message part of the port type operation, `reservation`, are constructed from the input parameters of the mediator service operation, `carReserv`. There is one `paramMap` element for each field in the input message parts of the selected port type operation within the enclosing the `inputMap` element.

The first `paramMap` element defines a parameter-mapping that has as target node the XPath expression `reservationSoapIn/reservation/pInput/pickupInfo/@dt` - defined by the value of the attribute `targetParam`. There is only one source node for this parameter-mapping, the one defined by the XPath expression `@pickupDate`. According to this parameter-mapping, the field `reservationSoapIn/reservation/pInput/pickupInfo/@dt` of the input message part, of the port type operation, `reservation`, is constructed from the input parameter `pickupDate` of the mediator service operation, `carReserv`. As both nodes of the parameter-mapping are of the same type, no mapping function is needed. Thus, the attribute `mapFunction` does not appear in this `paramMap` element.

The second `paramMap` element defines a parameter-mapping that has as target node the XPath expression `reservationSoapIn/reservation/pInput/`

pickupInfo/@location - defined by the value of the attribute targetParam. There are three source nodes for this parameter-mapping. The first source node is defined by the XPath expression pickupLocation/@state. The second source node is defined by the XPath expression pickupLocation/@city. The last source node is defined by the XPath expression pickupLocation/@office. According to this parameter-mapping, the field reservationSoapIn/reservation/pInput/pickupInfo/@location of the input message part, of the port type operation reservation, is constructed from three input parameters of the mediator service operation, carReserv: pickupLocation/@state, pickupLocation/@city, and pickupLocation/@office. The value of the attribute mapFunction indicates that the mediator function concatValues must be used to convert the source nodes into the target node. This function will perform a many-to-one mapping, converting the values of the three source nodes into one value for the target node. For example, the concatValues function may receive a set of values and perform the concatenation of these values using a pre-defined separator symbol between them. The other parameter-mappings of the selected inputMap element are not shown, but they follow the same pattern of the parameter-mappings just described. The complete example, with all parameter-mappings, can be found in Appendix 7.

The outputMap element of the reservMap element prescribes how the output parameters of the mediator service operation, carReserv, are constructed from the fields in the output message part of the port type operation, reservation. There is one paramMap element for each output parameter of the selected mediator service operation within the enclosing the outputMap element.

The first paramMap element defines a parameter-mapping that has as target node the XPath expression @reservationCode - defined by the value of the attribute targetParam. There is only one source node for this parameter-mapping, the one defined by the XPath expression reservationSoapOut/reservationResponse/@reservationResult. According to this parameter-mapping, the output parameter reservationCode of the mediator service operation, carReserv, is constructed from the field reservationSoapOut/reservationResponse/@reservationResult of the output message part, of the port type operation, reservation. As both nodes of the parameter-mapping are of the same type, no mapping function is needed. Thus, the

attribute mapFunction does not appear in this paramMap element. The other paramMap element defines a parameter-mapping that has as target node the XPath expression @company - defined by the value of the attribute targetParam. There is only one source node for this parameter-mapping. This source node is described by a sourceParam element where the attribute fixed appears with value BrazilCar. According to this parameter-mapping, the output parameter company, of the mediator service operation, carReserv, is not constructed from a field in the output message part of the port type reservation. Rather, when invoking this port type, it is always a constant with value BrazilCar.

The faultMap element of the reservMap element matches a fault message of the mediator service operation, carReserv, to the field, in the output message part of the port type operation, reservation. The paramMap element, within the enclosing outputMap element, defines a parameter-mapping that has as target node the XPath expression @ERROR_100 defined by the value of the attribute targetParam. The source node for this parameter-mapping is defined by the XPath expression reservationSoapOut/reservationResponse/@reservationResult, while the value NO_AV_CARS is assigned to its attribute responseValue. According to this parameter-mapping, when the remote service operation returns an output message carrying the value NO_AV_CARS in its field reservationSoapOut/ reservationResponse/@reservationResult then the mediator service operation carReserv will return a fault message with the value ERROR_100 as its errorCode attribute.

The last definition enclosed by the reservMap element is the content description for the port type operation, reservation. Recall from Section 5.2.4 that the content description of a remote service operation is defined regarding the input parameters of its related mediator service operation. In the example, there are two contentDescription elements. The first is defined for the input parameter preferredCompany/@company of the mediator service operation, carReserv. This content description specifies that the selected remote service is able to attend requests when the value of the parameter inputMsg/preferredCompany/@companyName, of the mediator service operation, carReserv, is within the domain of values {BrazilCar}. That is, the remote service rsBrazilCar can be used to serve car reservations only when the company

`rsBrazilCar` is within the list of preferred car rental companies. The other `contentDescription` element is defined for the input parameter `inputMsg/pickupLocation/@country` of the mediator service operation, `carReserv`. This content description specifies that the selected remote service is capable of answering requests when the value of the parameter `inputMsg/pickupLocation/@country`, of the mediator service operation, `carReserv`, is within the domain of values {`Brazil`}.

6. Mediator Service Composition

The mediator service operations, described in the previous chapter, can be combined to construct new services. To address this issue, the WSTL framework provides elements for describing compositions of mediator service operations. These elements specify the appropriate *transaction interaction pattern* of a collection of mediator service operations, in such way that the resulting composition describes how to achieve the goal of a particular business process. In WSTL, every composition defines a transactional unit of work.

A composition is built by describing how to utilize the capability provided by a collection of mediator service operations. WSTL designs compositions as specifications of the execution sequence of a set of mediator service operations. Execution sequences are specified by defining dependencies among those mediator service operations. Besides the specification of execution sequences, WSTL compositions allow the definition of specific transactional behavior. The transaction behavior specifies the level of atomicity and reliability of a given WSTL composition.

The traditional notion of atomicity states that all operations of a transaction must be treated as a single unit; either all the operations are executed, or none [47], [54]. In WSTL, the traditional all-or-nothing semantics of transactions is relaxed through the notion of mandatory operations. A mediator service operation within a WSTL composition can be identified as being mandatory or optional. Considering the successful termination state of a composition, all executed mandatory operations are required to terminate successfully, while optional operations may or may not succeed. This leads to a flexible (and user defined) notion of atomicity. The transactional behavior of WSTL compositions defines a single transaction that *may* not be executed in an atomic fashion. This happens when a composition has at least one mediator service operation marked as optional. In this case, the decision to commit the composition is made despite the termination state of the optional operation. The composition can commit even when the optional operation fails, leading to a non-atomic commit decision.

The reliability of a composition is defined through the notion of contingency operations. Contingency operations are mediator service operations executed when a given mediator service operation fails. A contingency operation can be specified for each composed mediator service operation. The specification of contingency operations builds alternative execution paths increasing the composition reliability as a whole.

WSTL supports the recursive composition of mediator service operations. Therefore, a mediator service composition can be used as a component of new compositions. The ability of recursive composition allows the reuse of existent compositions, facilitating the building of more complex transaction patterns.

In the next sections, we describe the WSTL elements used for specifying transactional compositions of mediator service operations.

6.1 REFERENCE MODEL FOR SPECIFYING COMPOSITIONS

WSTL models compositions as *composite tasks*. A composite task is represented by a labeled directed graph where nodes represent steps of execution and edges represent the flow of control and data among different steps. Each step of a composite task is either an atomic *task* or another composite task. An atomic task is a unit of work that is executed by a mediator service operation. Therefore, atomic tasks have a mediator service operation assigned to it, which is invoked when the task is executed.

Tasks are identified by a name and have: a signature, a set of execution dependencies, a set of data links, and, optionally, a set of rules.

The *signature* of an atomic task is related to the input, output, and fault messages of the mediator service operation that is used as the implementation of the task.

Execution dependencies are based on the execution state of other tasks defining the first kind of edges in the graph that represent a composite task. An execution dependency is defined among related tasks and it is a constraint on the temporal occurrence of the start and termination events of them. Execution dependencies define the order in which tasks must be executed, i.e., the composition control flow. An execution dependency is specified based on the *execution state* of one or more tasks.

Data links are mappings between messages belonging to the signatures of related tasks to allow the exchange of information between these tasks. Data links are the second kind of edges in the graph that represents a composite task.

Rules specify the conditions under which certain events will happen. Rules can be associated to dependencies or to data links. A dependency that has a rule will evaluate to true if both the dependency *and* the rule evaluate to true. A data link that has a rule will evaluate to true if the rule evaluates to true. Data links without explicit rules always evaluate to true.

Besides the components described above, composite tasks have a set of *mandatory tasks*. This set specifies the component tasks that must commit in order to commit the composite task. A user can specify a composite task aggregating tasks that are desirable, but not essential, to accomplish the target task. The set of mandatory tasks allows the distinction between *desirable* and *mandatory* tasks, providing more flexibility while specifying a composition. This flexibility increases the composition robustness, since the set of tasks required to commit, in order to commit the composition, are formed only by that tasks that are essential to accomplish the composition target. Thus, the composition will successfully terminate even if a subset of its component tasks fails, as long as all its mandatory tasks successfully commit.

6.1.1 Execution States of Tasks

A task can be in one of the following states: *not-executed*, *executing*, *committed*, *aborted*, *compensated*. The *not-executed* state models an inactive task, i.e., a task that has not been scheduled to execute. The *executing* state identifies running tasks, while the states *committed*, *aborted*, and *compensated* represent the possible terminate states of a task. Not all states are valid for all kind of tasks.

The valid states for atomic tasks are dependent on the valid states of the mediator service operations used for implementing the task. For tasks implemented by pivot operations (vide Section 3.1.2), only the states *not-executed*, *committed*, and *aborted* are valid, while for tasks implemented by compensable operations all states are valid. When a task is implemented by a retriable operation, only the states *not-executed*, *executing*, and *committed* are valid.

For composite tasks, the valid states are dependent on the valid states of each of the tasks comprising the composite task as well as the specification of the mandatory tasks of the composite task. If all component tasks, marked as mandatory, are compensable then the composite task itself will expose all possible states. If there is at least one component task marked as mandatory that is pivot, then the composite task will expose all states but the *compensated* state. The reasoning about the termination state of a composite task is described in the next chapter.

Transitions between the various states of a task are affected by different scheduling events. Some transitions are controlled by the scheduler responsible for enforcing inter-task dependencies. For example, the WebTransact system can submit a task for execution thus performing a state transition from state *not-executed* to state *executing*. Other transitions are controlled by the mediator service responsible for the execution of the task. For example, a running task may be unilaterally aborted by the mediator service, thus resulting in a state transition from *executing* to *aborted*.

6.1.2 Execution Dependencies

The goal of execution dependencies is to control the state transition of tasks in a composite task execution. Execution dependencies are the first type of directed edges in the graph that represent a composite task and define a *control link* between the tasks of a composite task. This control link has a *source* task and a *target* task associated to it. The state transition of the target task is dependent of some execution state of its related source task. An execution dependency has the form: $A \xrightarrow{(x,y)} B$; where A is the source task, B is the target task, and the label (x, y) is a state transition condition. In a state transition condition, x represents a set of execution states while y represents a command to perform a state transition. The commands to perform state transitions are: *start* and *compensate*. The *start* command performs a state transition from the state *not-executed* to the state *executing*. The *compensate* command performs a state transition from the state *committed* to the state *compensated*. An execution dependency $A \xrightarrow{(x,y)} B$ means: "If task A reaches state x then apply command y to task B". For example, consider that x holds the value $\{commit\}$ and that y is the command *start*. Then, the execution dependency above will evaluate to true when task A reaches the state *committed* triggering the execution of command *start* over task B. The *start* command will perform a state transition of task B from the state *not-executed* to the state *executing*.

There are six different types of execution state dependencies: *start-start*, *commit-start, abort-start*, *commit_abort-start*, *weak_commit-start*, and *commit-compensate_abort*.

The *start-start* dependency identifies tasks that do not have execution dependencies, i.e., the source task is null. Tasks that have only the *start-start* dependency are scheduled to run when the composition is instantiated by WebTransact.

The *commit-start* dependency synchronizes the execution starting point of the target task to the *committed* state of the source task while the *abort-start* dependency synchronizes the execution starting point of the target task to the *aborted* state of its source task.

The *commit_abort-start* dependency synchronizes the execution starting point of the target task to the *committed* or the *aborted* state of the source task. The *commit-compensate_abort* dependency indicates that the target task must be compensated or aborted if the source task reaches the state *committed*. These dependencies are used for modeling the execution of alternative tasks as described in Section 6.4.1. Alternative tasks define a set of tasks executed concurrently where only one of them can stay in the *committed* state after the whole set has terminated its execution. The *commit_abort-start* dependency is also used to model optional tasks. Since an optional task does not influence the final state of its composite task, all control links leaving such task must be labeled with *commit_abort-start* dependency.

The *weak_commit-start* dependency is also used for modeling the execution of alternative tasks. Differently from the other dependencies, the *weak_commit-start* dependency is applied to a set of control links having the same target task. This dependency synchronizes the execution starting point of the common target task to the *committed* state of the first source task to succeed. Therefore, the execution starting point of the target task is *not* tied to the *committed* state of all source tasks but only to the *committed* state of the first source task to commit.

All the above dependencies synchronize the execution starting point of the target task to the *committed* and/or *aborted* states of the source task(s). Therefore, these dependencies are all applied to target tasks whose state is none but the *not-executed* state.

The *commit-compensate_abort* dependency is the only dependency that does apply to target tasks already submitted for execution. The *commit-compensate_abort* dependency indicates that the target task must be aborted or compensated, depending on its current state, when the source task reaches the *committed* state.

From now on, we will be using the name *start* dependencies to identify the set of all dependencies that synchronizes the execution starting point of a given task while the *commit-compensate_abort* dependency will be referenced simply as *commit-compensate* dependency.

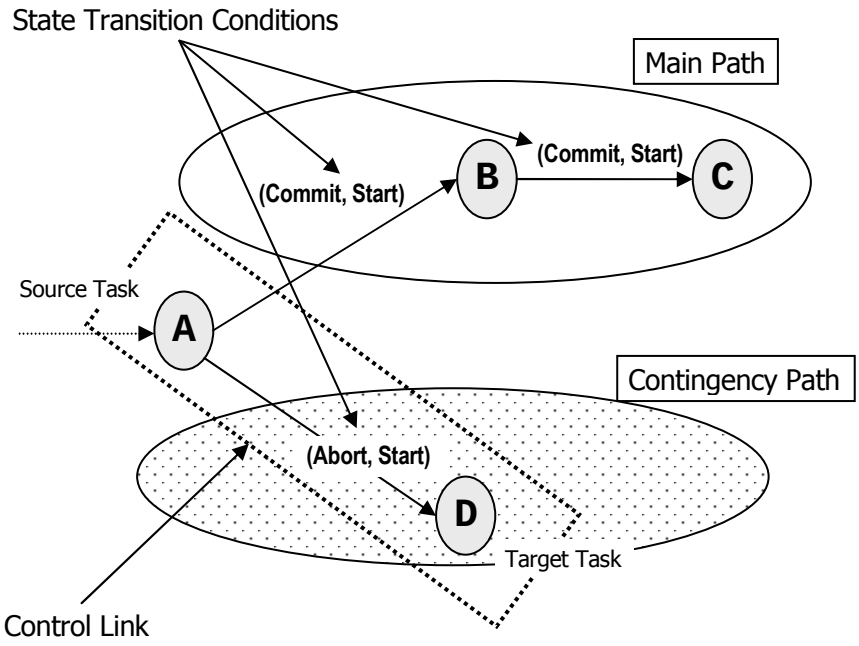


Figure 6.1 - Using an *abort-start* execution dependency to define a contingency execution path.

The *commit-start* dependency and *abort-start* dependency can be combined to build contingency execution paths. A contingency execution path defines a contingency plan for a composite task execution. A contingency plan is a set of tasks executed when a given task fails. For example, in Figure 6.1 tasks `B` and `D` have an execution dependency regarding task `A`. Task `B` has a *commit-start* dependency while task `D` has an *abort-start* dependency. These dependencies together generate a branch in the graph of its composite task. This branch defines a contingency execution path. Contingency execution paths allow the specification of custom recovery procedures, improving the reliability of the overall composition. When a task execution failure occurs, the composition can still

continue its execution, as long as a contingency execution path can be reached after the occurrence of the task failure.

6.1.3 Data Links

A *data link* defines the flow of data between the tasks of a composite task. A data link has a *source* task and a *target* task associated to it. The data flows from the source to the target, i.e., the source is the data producer and the target is the data consumer. Data links are the second type of directed edges in the graph that represents a composite task.

A data link can be specified only if the target of the data link is reachable from the source of the data link through a path of directed control links. The goal of this constraint is to avoid some incorrect specifications. For example, the specification of data links between a target task willing to consume data from a source task that has not finished its execution yet, or dead lock situations in which one task requires data from another task as input but the latter task needs the output of the former as its input. It is important to note that the target task of a data link is not necessarily the immediate successor of the task that produces the data being consumed. Many different tasks might be visited along the path defined by control link from the source of data link to the target of the data link.

A task might be the target of multiple data links. This allows aggregating input from multiple sources as well as specifying alternative input from tasks of alternative paths.

A data link can assign a part of a source message into a bound variable. The bound variable is in the scope of all tasks that reference the data link. A bound variable has no namespace associated with it, and it is referenced from an XPath expression using the prefix $ and the same name as specified by the name attribute.

6.1.4 Rules

A rule is a condition that can be used along execution dependencies or data links.

When used along a data link, rules are evaluated against the contents of messages or the execution state of the source task of the data link. Rule messages can be specified only if the target task of the data link is reachable from its source task, through a path of directed control links. The goal of this constraint is the same as in data links, i.e., to avoid

incorrect specifications. When a data link has a rule associated to it, its target task will consume the messages produced by its source task only if the rule evaluates to true.

When used along execution dependencies, rules are evaluated against any data in the scope of the selected execution dependency. The evaluation of a rule constrains which tasks are candidates for execution. Execution dependencies associated with rules are evaluated to true when both its state transition condition *and* the rule evaluate to true.

6.1.5 Mandatory Tasks

Mandatory tasks specify, for each execution path of a composite task, the set of tasks that must commit in order to commit the composite task. A user can specify a composite task aggregating desirable, but not essential, tasks to accomplish the task target. The set of mandatory tasks allows the distinction between *desirable* and *mandatory* operations, providing more flexibility while specifying the application. This flexibility improves the task robustness, since the set of tasks required to commit, in order to commit the composite task, are formed only by that tasks essential to accomplish the composite task target. Thus, the composite task will successfully terminate even if a subset of its component tasks fails, as long as all its mandatory tasks successfully commit.

The set of mandatory operations along with the contingency execution paths, define the *set of acceptable states* for the successful termination of a composite task. For each execution path, there is a non empty set of acceptable states defining the acceptable final states for each task of the composite task. If an acceptable state is reached during the scheduling of the composite task, no additional task need to be scheduled, and the composite task can terminate successfully. As an example, consider the composite task specified in Figure 6.2. The composite task is represented as a directed graph where the nodes are tasks and the edges are control links representing execution dependencies. There are two possible execution paths, *P1* = {A, B, C, D} and its contingency path, *P2* = {A, E, F}. Let the set of mandatory tasks be *M* = {A, B, C, D, E}. Then, we have two sets of acceptable states, one for each execution path. For the first execution path we have the set *AS1* = {[A- *committed*, B- *committed*, C- *committed*, D- *committed*]} and for the second execution path, which is a contingency path regarding task B, we have the set *AS2* = {[A- *committed*, E- *committed,* , F- *committed*] ; [A- *committed*, E- *committed,* , F- *aborted*]}. The composite task will successfully terminate if any acceptable state is reached. It is

important to note that mandatory tasks must succeed only when considering the execution path that they belong to. For example, in the composite task specified in Figure 6.2, task E is mandatory, but if the control flow goes through the execution path *P1*, this task will not be executed, so it will not be considered when deciding the commit of the composite task, even though it is marked as mandatory operation. Another case occurs when task B aborts. After the abort of task B the execution flows through the contingency path *P2*, thus despite the mandatory status of task B, it is not required to commit when considering the execution path *P2*.

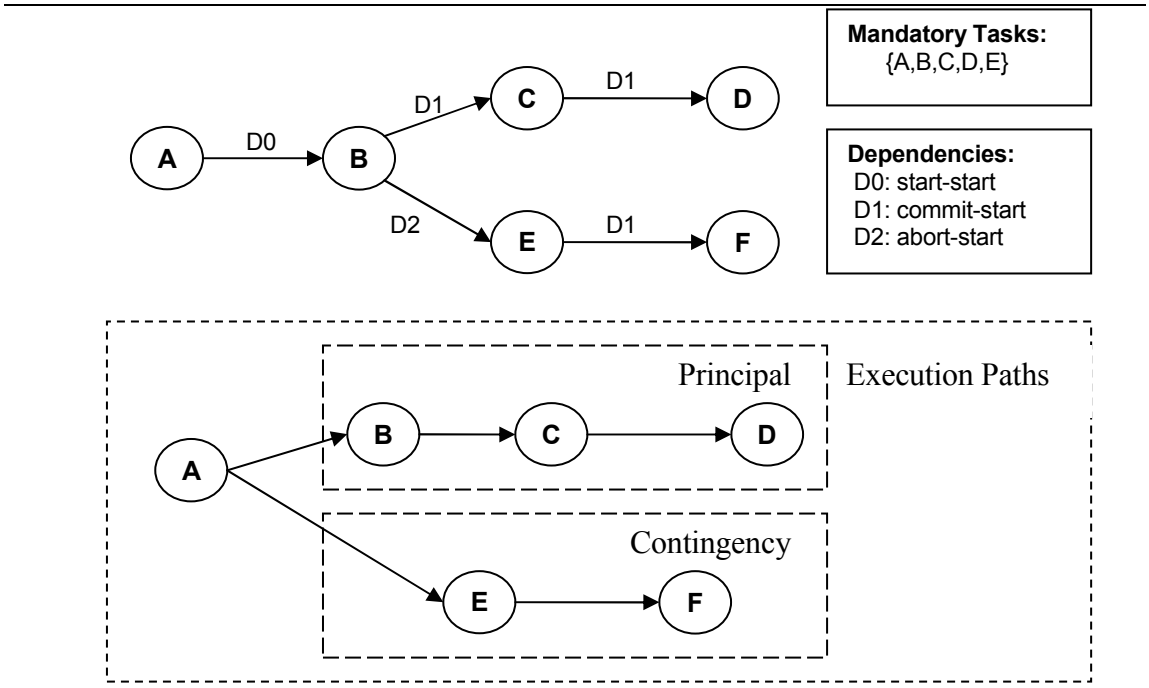


Figure 6.2 - Mandatory tasks definition example.

In this section, we have just highlighted the issue of acceptable termination states of a composite task. This issue will be analyzed in depth in Chapter 7.

6.2 COMPOSITE TASK EXAMPLE

To illustrate the features of WSTL to model compositions, we model a business transaction of planning a trip[8].

[8] The complete WSTL definitions for this example can be found in Appendix 10. Other examples can be found in [http://www.cos.ufrj.br/~pires/webTransact.html].

The trip plan consists of a hotel reservation and a car reservation for researchers attending a conference. The hotel and car reservations must be scheduled according to the following constraints. First, the system must try to make a hotel reservation in the conference hotel. If this reservation fails, the system must try a hotel reservation in any other hotel belonging to the list of indicated hotels of the conference organization. If this reservation succeeds, the system must try a car reservation, but only if the total price of the hotel room was less than a fixed value. The trip plan will succeed if any hotel reservation succeeds despite of the car reservation result.

The trip plan described previously involves three different tasks:

1) An atomic task for booking the conference hotel implemented by a mediator service operation for booking hotel rooms;

2) An atomic task for booking a hotel other than the conference hotel implemented by the same mediator service operation of item 1;

3) An atomic task to make the car reservation implemented by a mediator service that makes car reservations; and

4) A composite task for implementing the composition of the above tasks. This composite task implements the trip plan as a transactional process that is executed by the WebTransact system.

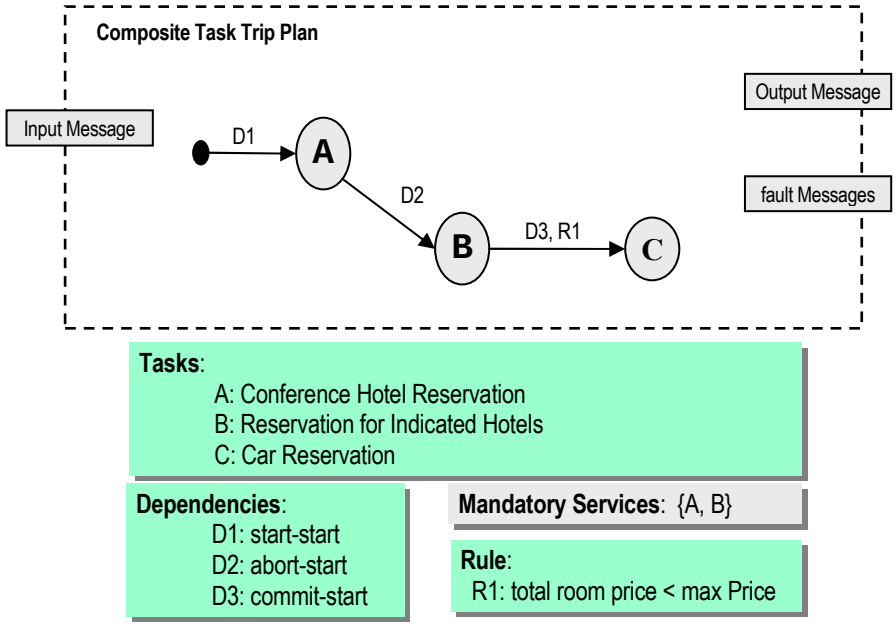


Figure 6.3 - A directed graph representing the trip plan task. The nodes are the atomic tasks for booking a hotel room and for reserving a car. The edges are the control links. Each control link has a label representing the state transition condition as well as its associated rules.

Figure 6.3 shows a directed graph representing the trip plan task. The graph shows the atomic tasks and its execution dependencies. Figure 6.4 shows another directed graph representing the trip plan task. Now, the graph represents the data links that specifies the flow of data between the tasks.

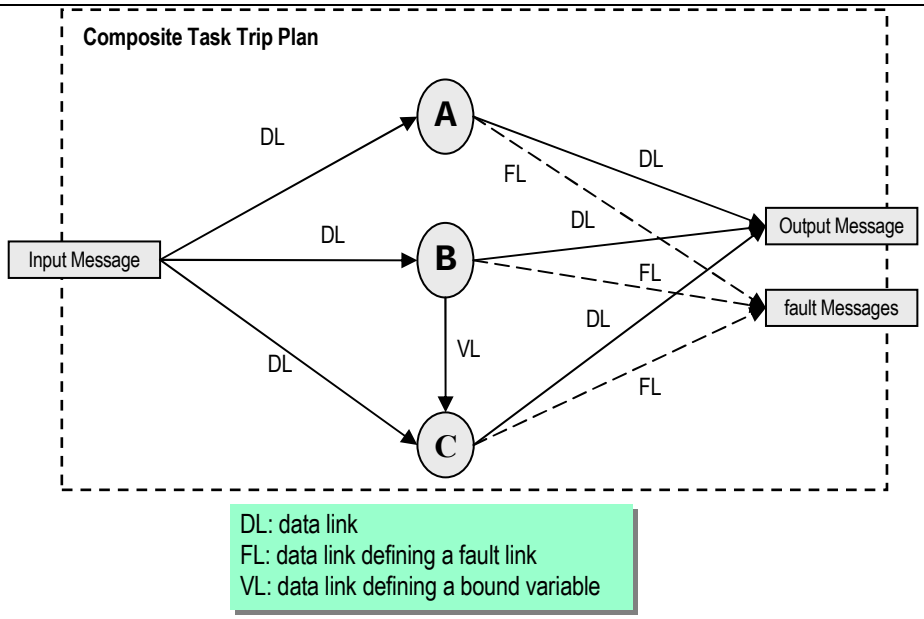


Figure 6.4 - Another directed graph representing the trip plan task. This graph shows the edges as data links.

We use the namespace prefix `mstrip` to denote the business domain of the hotel and car mediator services, the namespace prefix `mstripxsd` to denote the type definitions of the hotel and car mediator services; the namespace prefix `tsktp` to denote the business domain of composite task trip plan; and the namespace prefix `lxsd` to denote the type definitions of the composite task trip plan.

6.2.1 The Trip Plan Composite Task

We start by defining the composite task representing the trip plan. The first step is the definition of the input, output and fault messages of the composite task. Input messages are the container used by composite tasks for communicating data to its component tasks. Output and fault messages are the containers used by composite tasks for communicating data from its component tasks to the consumer of the composite task. The consumer of a composite task can be another composite task or an application program. Messages are defined using XML Schema, like all other definitions in WSTL.

The following document fragment illustrates how to define the composite task that implements the trip plan using WSTL.

```
<definitions ...

  <types>
    <schema
      targetNamespace="http://example.com/Applications/TripPlan/Schema/"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType   name="tripPlanType">
        <sequence>
          <element name="Person" type="mstripxsd:PersonType"/>
          <element name="originInfo" type="mstripxsd:infoType"/>
          <element name="destinationInfo" type="mstripxsd:infoType"/>
          <element name="preferredAirlineCompanies"
                type="mstripxsd:companiesType" use="optional"/>
          <element name="indicatedHotels" type="indicatedHotelsType"/>
        </sequence>
        <attribute   name="conferenceHotel" type="string" use="optional"/>
        <attribute   name="maxPriceToCarReserv" type="string" use="optional"/>
      </complexType>
      <complexType   name="tripPlanResponseType">
        <sequence>
          <element name="carResponse"
                type="mstripxsd:carReservationInfoType" minOccurs="0"/>
          <element name="hotelResponse" type="mstripxsd:hotelReservationInfoType"/>
        </sequence>
      </complexType>
      <complexType   name="indicatedHotelsType">
        <sequence>
          <element name="hotelName" type="string"/>
        </sequence>
      </complexType>
    </schema>
  </types>

  <wstl:compositionDefinition>

<!-- ++++++++++++++++++       Composite task Trip Plan ++++++++++++++++++++++++++++++ -->
    <wstl:compositeTask id="tripPlan">
      <wstl:dependency type="start-start"/>
```

```
      <wstl:messages>
        <wstl:inputMsg>
          <wstl:param name="tripInfo" type="lxsd:tripPlanType"/>
        </wstl:inputMsg>
        <wstl:outputMsg>
          <wstl:param name="tripPlanResponse" type="lxsd:tripPlanResponseType"/>
        </wstl:outputMsg>
        <wstl:faultMsg errorCode="ERROR_500" description="No available rooms"/>
        <wstl:faultMsg errorCode="ERROR_501" description="No available transportation"/>
        <wstl:faultMsg errorCode="ERROR_102" description="Communication failure"/>
      </wstl:messages>
      ...


    </wstl:compositeTask>

  </wstl:compositionDefinition>
</definitions>
```

Figure 6.5 - The composite task for the trip plan transaction process.

The root definition for specifying WSTL compositions is the composition definition element. This element may have many composite tasks definitions inside it. In Figure 6.5 one composite task is specified, named `tripPlan`. This composite task has one input message, one output message, and three fault messages. These messages are defined based on the messages of the tasks that compose the composite task. For example, consider the input parameter `tripInfo` of type `tripPlanType` . All the elements of the `tripPlanType` type are defined based on the XML schema of the mediator services that implements the component tasks: car reservation, and hotel reservation. A parameter of an input message can also be defined based on a specific input data used for constraining the execution semantics of the composite task. For example, the attribute `indicatedHotels` - of type `indicatedHotelsType` - is defined in the local XML schema of the composition itself. This attribute is related to the semantics of the composition rather than the input messages of one of its component tasks.

6.2.2 The Component Tasks

The next step is the specification of the atomic tasks that compose the composite task `tripPlan`. Atomic tasks are implemented by a mediator service operation and have a name, a signature, a set of execution dependencies, a set of data links, and optionally, a set of rules.

6.2.2.1 Atomic Task for Booking a Room in the Conference Hotel

The following XML fragment illustrates how to define the atomic task for booking a room in the conference hotel using WSTL.

```
<wstl:atomicTask  id="conferenceHotelReservation"
                  medService="msTrip:msHotelReservation"
                  operation="hotelReserv">
  <wstl:dependency type="start-start"/>
  <wstl:dataLinkRef name="dlHotelInfo"/>
  <wstl:dataLinkRef name="dlConferenceHotel"/>
</wstl:atomicTask>
```

Figure 6.6 - Definition of the atomic task conference Hotel reservation.

The value of the attribute `id - conferenceHotelReservation -` uniquely identifies the task inside a WSTL document. The mediator service operation that implements the task is defined in the value of the attribute `operation`. The definition of this operation is specified in the value of the attribute `medService`. The signature of the atomic task is the signature of its associated mediator service operation. This is implicitly defined for atomic task definitions.

Figure 6.3 shows a directed graph representing the execution dependencies of the trip plan task. In that figure, there is only one directed control link flowing to the task Conference Hotel Reservation. This control link is defined by a dependency that has a *start-start* state transition condition and an empty task as its source task. Therefore, the Conference Hotel Reservation task will be scheduled to run when the composite task is instantiated.

Two *data link references* are attached to the Conference Hotel Reservation task. These data link references indicate the data links that must be performed before the starting of such task.

The XML fragment in Figure 6.7 shows the definitions of the data links associated with the Conference Hotel Reservation task using WSTL.

```
<wstl:dataLink name="dlHotelInfo" source="tsktp:tripPlan">
  <wstl:link   target="inputMsg/param[name='responsible']"
               selection="inputMsg/param[name='tripInfo']/Person"/>
  <wstl:link   target="inputMsg/param[name='checkinDate']"
               selection="inputMsg/param[name='tripInfo']
                         /originInfo/schedule/@date"/>
  <wstl:link   target="inputMsg/param[name='checkoutDate']"
               selection="inputMsg/param[name='tripInfo']
                        /destinationInfo/schedule/@date"/>
  <wstl:link   target="inputMsg/param[name='location']"
               selection=”inputMsg/param[name='tripInfo']
                        /destinationInfo/@location"/>
</wstl:dataLink>
<wstl:dataLink name="dlConferenceHotel" source="tsktp:tripPlan">
  <wstl:link   target="inputMsg/param[name='preferredHotels']/companyName[1]"
               selection="inputMsg/param[name='conferenceHotel']"/>
```

```
</wstl:dataLink>
```

Figure 6.7 - Definition of data links associated with the atomic task Conference Hotel Reservation.

These data links define the flow of data that must be assigned before the task Conference Hotel Reservation is started. Recall from Section 6.1.3 that each data link has a source and a target task. The source task is defined by the attribute `source` of the data link element while the target task is defined by the context in which the data link is used. For example, the data link named `dlHotelInfo` (Figure 6.7) has the composite task Trip Plan as its source task. The target task of this data link is defined depending on the context where the data link is used. This context is set when the data link is referenced in a task definition. As the task Conference Hotel Reservation references the `dlHotelInfo` data link, it establishes a context for that data link. Therefore, in this context, the `dlHotelInfo` data link has the task Conference Hotel Reservation task as its target task and the composite task Trip Plan as its source task. The target and source tasks of a data link define a source context and a target context used by the WebTransact system to perform the flow of data.

A data link can have a set of *link* child elements. Each link element defines a flow of data from parts of messages elements of the source context to parts of messages elements of the target context. A link has a target and a selection attribute. The target attribute creates a new target context based on the target context established by the target task of the data link. The select attribute is an XPath expression that operates on the source context and returns an XPath result. This result is assigned by the WebTransact system to the current target context. For example, the first link of the `dlHotelInfo` data link (Figure 6.7) has the following source and target contexts:

```
Source context = "tsktp:tripPlan"
Target Context = "tsktp:conferenceHotelReservation/inputMsg/param[name='responsible']"
```

Therefore, the selection `"inputMsg/param[name='tripInfo']/Person"` will return an element containing the `Person` element of the parameter named `tripInfo` of the input message of the `tripPlan` task. The returned element will be attached to the parameter named `responsible` of the input message of the `conferenceHotelReservation` task.

The remaining links of the `dlHotelInfo` data link are built in the same way as the link described above. The other data link - `dlConferenceHotel` - that appears in Figure 6.7 is built similarly to the `dlHotelInfo` data link.

6.2.2.2 Atomic Task for Booking a Room in Any of the Conference Indicated Hotels

The following XML fragment illustrates how to define the atomic task for booking a room in some hotel indicated by the conference organization using WSTL.

```
<wstl:atomicTask  id=" anyHotelReservation "
                  medService="msTrip:msHotelReservation"
                  operation="hotelReserv">
  <wstl:dependency type="abort-start" source="tsktp:conferenceHotelReservation"/>
  <wstl:dataLinkRef name="dlHotelInfo"/>
  <wstl:dataLinkRef name="dlHotel"/>
</wstl:atomicTask>
```

Figure 6.8 - Definition of the atomic task Any Hotel Reservation.

The value of the attribute `id` - `anyHotelReservation` - uniquely identifies the task inside a WSTL document. The mediator service operation that implements the task is the same operation used for implementing the `conferenceHotelReservation` task.

Figure 6.3 shows only one directed control link flowing to the task Any Hotel Reservation. This control link is defined by a dependency that has an *abort-start* state transition condition and the Conference Hotel Reservation task as its source task. Therefore, the Any Hotel Reservation task will be scheduled to run only if the Conference Hotel Reservation task fails.

Two *data link references* are attached to the Conference Hotel Reservation task. These data link references indicate the data links that must be performed before that task is scheduled to run. The `dlHotelInfo` data link is the same data link referenced in the definition of the Conference Hotel Reservation task (Figure 6.7). The XML fragment in Figure 6.9 shows the definitions of the `dlAnyHotel` data link.

```
<wstl:dataLink name="dlAnyHotel" source="tsktp:tripPlan">
  <wstl:link   target="inputMsg/param[name='preferredHotel']"
               selection="inputMsg/param[name='indicatedHotels']" />
</wstl:dataLink>
```

Figure 6.9 - Definition of data links associated with the atomic task Conference Hotel Reservation.

The data link `dlHotelInfo` is reused for the Any Hotel Reservation task. This is the reason of defining two different data links for the Conference Hotel Reservation task with the same source task. The `dlHotelInfo` data link defines common data that flow from the Trip Plan task to both the Conference and the Any Hotel Reservation atomic tasks. The other data links - `dlConferenceHotel` and `dlAnyHotel` - define the specific data flow to each one of the atomic tasks. Now, the target context of the `dlHotelInfo` data link is the Any Hotel Conference.

6.2.2.3 Atomic Task to Make Car Reservation

Having defined the atomic tasks for booking a hotel room, the next step is the definition of the atomic task to make a car reservation. The following XML fragment shows the definition of the atomic task that makes a car reservation using WSTL.

```
<wstl:atomicTask  id="carReservation"
                  medService="msTrip:msCarReservation"
                  operation="carReserv">
  <wstl:dependency type="commit-start" source="tsktp:anyHotelReservation">
    <wstl:rule condition="$hotelRoomPrice &lt; $maxPrice"/>
  </wstl:dependency>
  <wstl:dataLinkRef name="dlCarInfo"/>
  <wstl:dataLinkRef name="dlHotelRoomPrice"/>
</wstl:atomicTask>
```

Figure 6.10 - Definition of the atomic task Car Reservation.

The composite task Trip Plan defines one control link flowing to the Car Reservation task (Figure 6.3). This control link is defined by an execution dependency that has a *commit-start* state transition condition and the Any Hotel Reservation task as its source task. This dependency has a rule associate to it. When a dependency has a rule associated to it, its state transition is executed only if that rule evaluates to true. In this example, the rule condition is defined over two bound variables - `hotelRoomPrice` and `maxPrice`. Recall from Section 6.1.3 that bound variables are specified by variable link elements, which are defined inside data link elements. Figure 6.11 shows the definition of the data links referenced by the execution dependency of the Car Reservation task.

```
<wstl:dataLink name="dlCarInfo" source="tsktp:tripPlan">
  <wstl:link target="inputMsg/param[name='person']"
             selection="inputMsg/param[name='tripInfo']/Person">
  </wstl:link>
      ...

  <wstl:variableLink   name="maxPrice">
    <wstl:link   selection="inputMsg/param[name='tripInfo']/@maxPriceToCarReserv"/>
  </wstl:variableLink>
</wstl:dataLink>
<wstl:dataLink name="dlHotelRoomPrice" source="tsktp:anyHotelReservation">
  <wstl:variableLink   name="hotelRoomPrice">
```

```
    <wstl:link selection="outputMsg/param[name='info']/@price"/>
  </wstl:variableLink>
</wstl:dataLink>
```

Figure 6.11 - Fragment of the definition of data links associated with the atomic task Car Reservation.

The `maxPrice` bound variable is defined by a variable link inside the `dlCarInfo` data link while the `hotelRoomPrice` bound variable is defined by another variable link inside the `dlHotelRoomPrice` data link. A variable link creates a new target context and assigns the result of its nested data assignments to this context. The target context is then bound to the `name` attribute of the variable link. The source context of the nested data assignments is evaluated in the same way as for links (Section 6.2.2.1). For example, the variable link defined in the `dlCarInfo` data link has the Trip Plan task as its source context thus the nested assignment defined by the selection "`inputMsg/param[name='tripInfo']/@maxPriceToCarReserv`" will return the value of the attribute `maxPriceToCarReserv` of the parameter named `tripInfo` of the input message element of the Trip Plan task. The returned value will be bound to the variable named `maxPrice`. The same process occurs to bind the value of the variable `hotelRoomPrice`.

After the data links `dlCarInfo` and `hotelRoomPrice` are assigned, the WebTransact system will evaluate the rule associated with the execution dependency of the Car Reservation task. This rule states that the value of the bound variable `hotelRoomPrice` must be lower than the value of the bound variable `maxPrice`. The Car Reservation task will be scheduled to run only if this rule evaluates to true. Therefore, the execution of the Car Reservation task is dependent on the execution state of the Any Hotel Reservation task as well as the value of an output message part of this task.

6.2.3 Revisiting the Trip Plan Composite Task

Having all the atomic tasks that compose the Trip Plan task defined, we can finish its definition.

The XML Schema fragment in Figure 6.12 shows the complete definition of the composite task Trip Plan. All elements of such definition are explained as follows.

```
<wstl:compositeTask id="tripPlan">
     <wstl:dependency type="start-start"/>
```

```
      <wstl:messages
          <!-- Definitions in Figure 6.5      -->
      </wstl:messages>
      <wstl:taskRef name="tsktp:carReservation"/>
      <wstl:taskRef name="tsktp:conferenceHotelReservation"/>
      <wstl:taskRef name="tsktp:anyHotelReservation"/>
      <wstl:mandatoryTask>
        <wstl:taskRef name="tsktp:conferenceHotelReservation"/>
        <wstl:taskRef name="tsktp:anyHotelReservation"/>
      </wstl:mandatoryTask>
      <wstl:dataLinkRef name="dlConferenceHotelReservation"/>
      <wstl:dataLinkRef name="dlAnyHotelReservation"/>
      <wstl:dataLinkRef name="dlCarReservation"/>
      <wstl:faultLinkRef name="flConferenceHotelReservation"/>
      <wstl:faultLinkRef name="flAnyHotelReservation"/>
      <wstl:faultLinkRef name="flCarReservation"/>
</wstl:atomicTask>
```

Figure 6.12 - Definition of the atomic task Conference Hotel Reservation.

We will start this final step assigning the atomic tasks defined in the previous sections to the composite task Trip Plan. This assignment is done through the `taskRef` element. There must be one element `taskRef` for assigning each one of the atomic tasks described in the previous section to the Trip Plan task. This assignment is necessary because composite tasks might be composed of other composite tasks. When a WSTL document contains the definition of this kind of task composition, it is necessary to specify the scope of each task of the document.

The next step is the definition of the mandatory tasks according to the transactional semantics of the composite task. For the Trip Plan task, the mandatory tasks are the tasks: Conference Hotel Reservation and the Any Hotel Reservation. This definition is done inserting `taskRef` elements referencing these tasks inside the `mandatoryTask` element.

The final step is the definition of the data links that will produce the output or the fault messages of the composite task.

The XML fragment in Figure 6.13 shows the definitions of the data links associated with the Trip Plan task using WSTL.

```
<wstl:dataLink  name="dlConferenceHotelReservation"
                source="tsktp:conferenceHotelReservation">
  <wstl:link target="outputMsg/param[name='tripPlanResponse']/hotelResponse"
             selection="outputMsg/param[name='reservationInfo']"/>
  <wstl:rule condition="committed(tsktp:conferenceHotelReservation)"/>
</wstl:dataLink>
<wstl:dataLink  name="dlAnyHotelReservation"
                source="tsktp:anyHotelReservation">
  <wstl:link target="outputMsg/param[name='tripPlanResponse']/hotelResponse"
              selection="outputMsg/param[name='reservationInfo']"/>
  <wstl:rule condition="committed(tsktp:anyHotelReservation)"/>
</wstl:dataLink>
<wstl:dataLink  name="dlCarReservation"
                source="tsktp:carReservation">
  <wstl:link   target="outputMsg/param[name='tripPlanResponse']/carResponse"
```

```
              selection="outputMsg/param[name='reservationInfo']"/>
  <wstl:rule condition="committed(tsktp:carReservation)"/>
</wstl:dataLink>
<wstl:faultDataLink name="flAnyHotelReservation"
                    source="tsktp:anyHotelReservation">
  <wstl:link target="faultMsg/param[errorCode='ERROR_500']"
             selection="faultMsg/param[errorCode='ERROR_105']"/>
  <wstl:rule condition="aborted(tsktp:anyHotelReservation)"/>
</wstl:faultDataLink>
<wstl:faultDataLink name="flConferenceHotelReservation"
                    source="tsktp:conferenceHotelReservation">
  <wstl:link   target="faultMsg/param[errorCode='ERROR_500']"
               selection="faultMsg/param[errorCode='ERROR_105']"/>
  <wstl:rule condition="aborted(tsktp:ConferenceHotelReservation)"/>
</wstl:faultDataLink>
```

Figure 6.13 - Definition of data links associated with the atomic task Conference Hotel Reservation.

The data links of Figure 6.13 define data flows from the output messages of one atomic task to a field of the output message of the composite task. Each data link has a rule associated to it. A data link is activated only if its associated rule evaluates to true. For example, the data link `dlConferenceHotelReservation` will be activated only if the state of the task Conference Hotel Reservation is *committed* at the time of the link evaluation. Data links of composite tasks are evaluated when all its component tasks have terminated its execution or when there are no more component tasks to be scheduled.

Figure 6.13 has the definition of a special kind of data link, the *fault link*. Fault links have the exactly same definition of regular data links but define flow of error messages instead of regular data. Task errors with the same error code are automatically bound. It is not necessary to define a fault link in these cases. For example, the fault message with error code = `"ERROR_102"` is automatically routed between any atomic task and the composite task Trip Plan because the error code is the same for the atomic tasks and the composite task.

6.3 COMPOSITION CONSTRUCTORS

In Section 6.2 we have shown the specification of a composite task for implementing a transactional process for planning a trip. According to the execution dependencies defined for each component task of the Trip Plan task, when the task is instantiated, its execution flows as follows.

The first task to be executed is the Conference Hotel Reservation task. If this task fails then the control will flow to the Any Hotel Reservation task. On the other hand, if

the Conference Hotel Reservation task succeeds no more tasks are scheduled to run. If the execution control flow to the Any Hotel Reservation task and it terminates successfully then the execution control *may* flow to task Car Reservation. The Car Reservation task is executed only if the rule assigned to its execution dependency evaluates to true. The execution flow just described defines three types of control flow constructors: the *sequence*, the *contingency*, and the *conditional* constructors (Figure 6.14). These constructors are only a subset of the set of the constructors that can be defined using WSTL. The definitions of execution dependencies and rules provide a powerful mechanism for specifying the execution control flow of composite tasks. In this section, we describe how dependencies and rules can be combined to generate different types of control flow constructors.

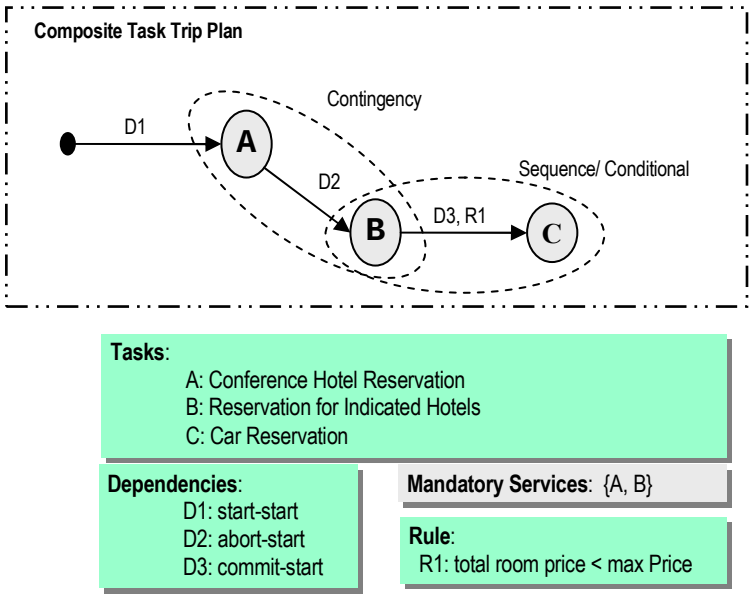


Figure 6.14 - Compositional constructor used for specifying the composite task Trip Plan.

There are six base types of control flow constructors in WSTL: *sequence*, *contingency*, *conditional*, *parallel*, *alternative*, and *iteration*.

The *sequence* constructor is defined by a control link labeled with a *commit-start* state transition condition, i.e., a *commit-start* execution dependency. The sequence constructor indicates that the execution of a composite task must flow from the source

task to the target task when the source task reaches the state *committed*. Figure 6.15 shows a graph fragment representing the sequence constructor.

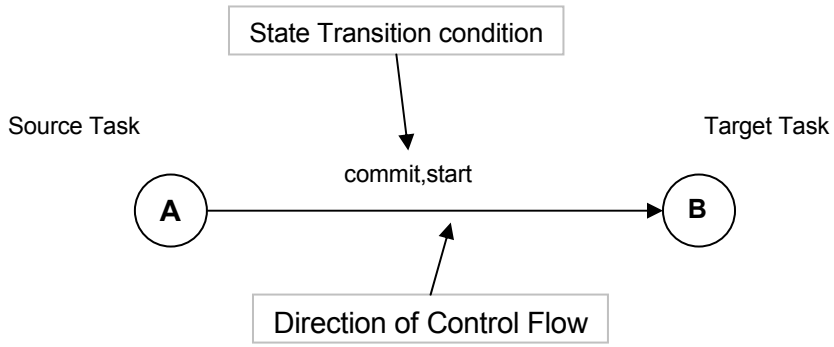


Figure 6.15 - The sequence constructor.

The *contingency* constructor is defined by a control link labeled with an *abort-start* state transition condition, i.e., an *abort-start* execution dependency. The contingency constructor indicates that the execution of a composite task must flow from the source task to the target task when the source task reaches the state *aborted*. Structurally, the contingency construct is similar to the sequence constructor as both constructors lead to an execution that flows from the source task to the target task of a given control link. However, they are semantically different. Setting the execution context over the source task, the sequence constructor specifies the main path of execution of a composite task.

On the other hand, the contingency constructor specifies a contingency path of execution for the selected composite task. Therefore, the combination of two control links having the same source task and different target tasks and being one a contingency constructor and the other a sequence constructor generates a branch in the execution flow of its composite task. This branch defines the beginning of a contingency path that has its context defined by the source task of both control links. Figure 6.15 shows a graph fragment representing this kind of branching.

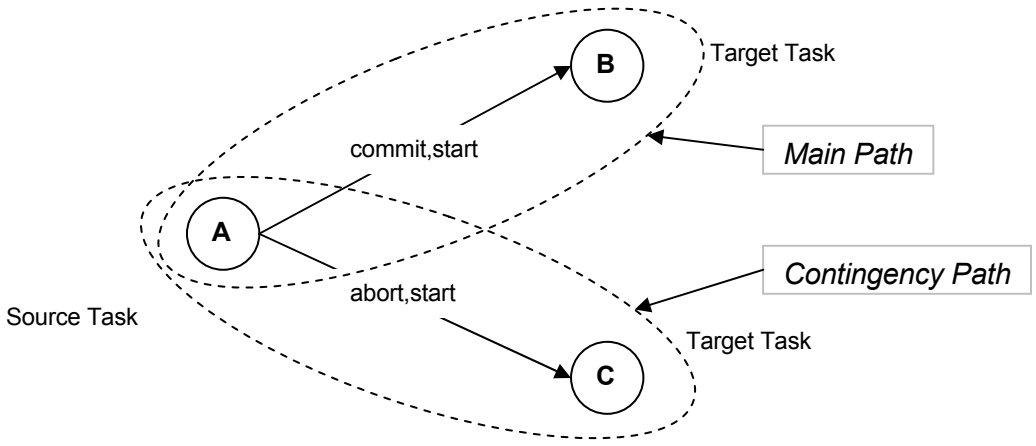


Figure 6.16 - The specification of a contingency execution path using the sequence and the contingency constructors.

The *conditional* constructor is defined by a control link labeled with an *abort-start* or *commit-start* state transition conditions and a rule, i.e., an *abort-start* or *commit-start* execution dependency associated with a rule. The *conditional* constructor indicates that the execution of a composite task must flow from the source task to the target task when the source task reaches the state specified by its labeled state transition condition *and* the rule associated to it evaluates to true. The conditional constructors are used for specifying branches in the execution flow of composite tasks dependent on execution states or values produced by previous executed tasks. Figure 6.15 shows a graph fragment representing this kind of branching. There are two control links in Figure 6.15: one has task `A` as its source task and task `B` as its target task; the other one also has task `A` as its source task and task `C` as its target task. Both control links are labeled with the *commit-start* state transition condition. However, only one of these control links is allowed to be activated during its composite task execution. What prevents the execution of both control links are the rules associated to them. The control link `A→B` is labeled with rule `R1` while control link `A→C` is labeled with the negation of the same rule.

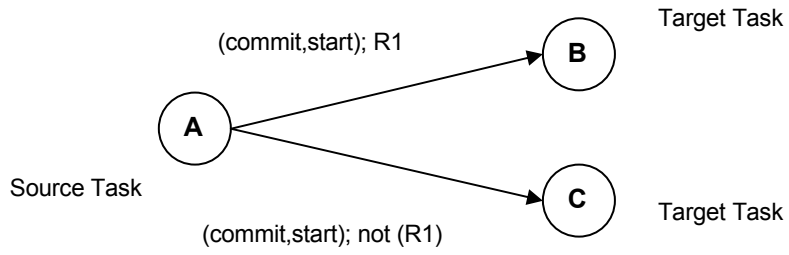


Figure 6.17 - The specification of the conditional constructor.

It is worth noting that nothing precludes the usage of different rules associated with pairs of control links defining conditional constructors. The control links of Figure 6.17 could have different rules associated to them instead of the same rule. In this case, it is not possible to infer in advance how it will be the execution control flow after the commit of task A. The execution control could flow to task A, task B or even both.

The *parallel* constructor is defined by a set of control links labeled with the *commit-start*, *abort-start* or the *start-start* state transition condition and having the same task as their source task. When the control links are labeled with the *start-start* state transition condition, the source task is the null task. This case models the control flow of composite tasks that have more than one task that must be executed when the composite task is instantiated. The parallel constructor indicates that the execution of a composite task must flow from the source task to all target tasks of its control links when the source task reaches the state specified by its labeled state transition condition. Figure 6.15 shows a graph fragment representing an example of the parallel constructor. When the task A reaches the execution state *committed*, tasks B and C will be executed concurrently.

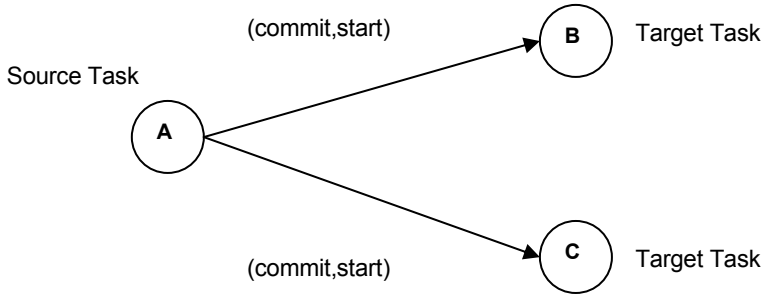


Figure 6.18 - The specification of the parallel constructor.

The alternative and the iteration constructor are used together with a special kind of task, named synchronization task, described in the next section.

6.4 THE SYNCHRONIZATION TASK

Besides the simple and composite tasks, WSTL makes use of a help task named *synchronization task*. The synchronization task is used for synchronizing concurrently executed tasks.

Synchronization tasks are like atomic tasks except for two things: there is no mediator service operation assigned to them and their signature has only output and fault messages.

A synchronization task is always associated with a set of different tasks. This set defines the *synchronized* tasks of the synchronization task. The association between a synchronization task and its synchronized tasks is defined through execution dependencies specifications. These execution dependencies specifications are represented by control links having a common target task, the synchronization task, and different source tasks, the synchronized tasks.

The possible execution states for synchronization tasks are: *not-executed*, *committed*, and *compensated*. The execution states *executing* and *aborted* do not apply to synchronization tasks since they do not have business processes associated to them. When a synchronization task is ready to run its state changes from *not-executing* to *committed*.

The valid execution states of synchronization tasks are dependent on the execution states of its synchronized tasks. If all synchronized tasks are *non*-compensable then the synchronization task will expose all states but the *compensated* state. If there is, at least one compensable synchronized task then the synchronization task itself will expose all possible states.

6.4.1 Synchronizing Concurrently Executing Tasks

In this section, we describe the usage of synchronization tasks for synchronizing concurrently executing tasks.

WSTL allows the specification of a special type of parallel execution named alternative parallel execution, alternative execution for short. The alternative execution models the concurrent execution of a set of tasks where only one of them can stay in the *committed* state after the whole set has terminated its execution. This type of execution is useful to improve the response time of composite tasks. If a user wants to model a composition having a set of semantically equivalent tasks, it is possible to execute these tasks in parallel and arbitrarily choose one of those that succeeded, say the first to succeed, and abort the others. This behavior is semantically equivalent to a composition

built using contingency constructors but it is more efficient regarding to response time. The alternative execution is modeled using the *alternative* constructor.

The *alternative* constructor is defined by a set of control links labeled with the *commit_abort-start*, or *weak_commit-start* state transition condition and having the same task as their target task. The task that acts as target task is always a synchronization task. The flow of control through an alternative constructor depends on the state transition condition labeled in the control links that composes it.

When the state transition condition is the *weak_commit-start*, the execution control will flow to the target task when the first source task reaches the *committed* state. All source tasks, but the first one to commit, are automatically aborted by the WebTransact system. Figure 6.19 shows a graph fragment representing an example of the alternative constructor. The task A will be executed when the first task between tasks B and C reaches the execution state *committed*.

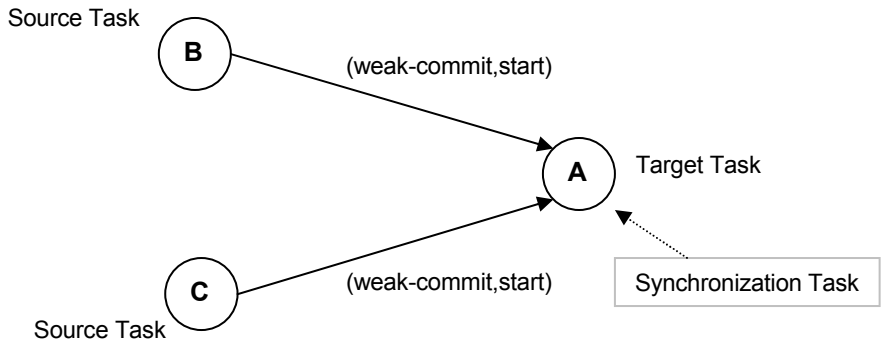


Figure 6.19 - The specification of the alternative constructor using the *weak_commit-start* dependency.

When the state transition condition is the *commit_abort-start* then the alternative constructor indicates that the execution must flow from the source tasks to the target task when all source tasks reach the *committed* or the *aborted* state. In this case, the synchronization task is a synchronization execution point for all source tasks. Alternative constructors defined by *commit_abort-start* dependencies need more information to be useful. Having in mind that we are defining alternative tasks, if more than one source task has committed, we need to specify which one of these tasks we really want to keep and those that must be discarded. Therefore, this type of alternative constructor is always

specified along control links elements flowing in the opposite direction, i.e., from the synchronization tasks to the source tasks. These control links are labeled with *commit-compensate* state transition conditions as well as rules. Therefore, they define the rules for compensating the tasks not desired. Figure 6.20 shows a graph fragment representing an example of the alternative constructor using the *commit-start* dependency. The execution control will flow to task `A` only after both tasks `B` and `C` reach the execution state *committed* or *aborted*. When this occurs, all data links associated with synchronization task `A` will be evaluated. Those data links having evaluated to true will be activated, i.e., data will flow from the source message of the data link to the associated message of the synchronization task. After that, if at least one of the synchronized tasks has committed, the synchronization task would change its state from *not-executed* to *committed*. On the other hand, if all synchronized tasks have aborted, the synchronization task would change its state from *not-executed* to *aborted*.

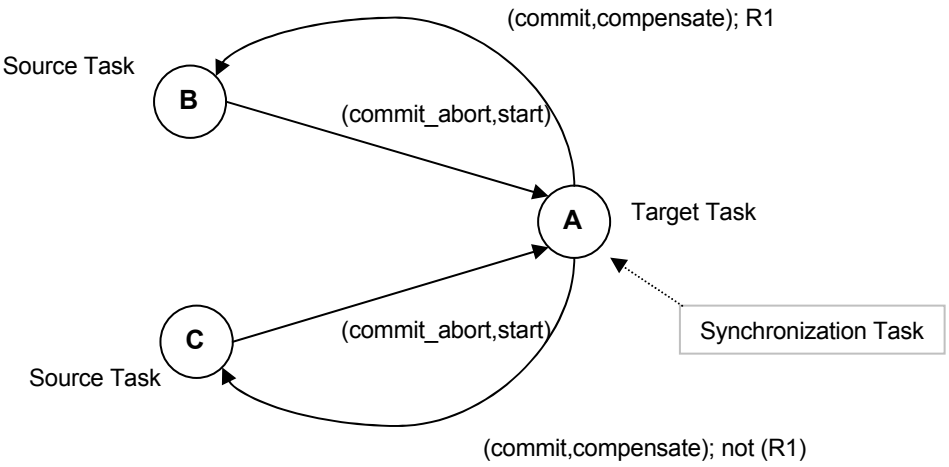


Figure 6.20 - The specification of the alternative constructor using the *commit_abort-start* dependency.

Figure 6.20 shows a cycle in the directed graph generated by the implementation of the alternative constructor. It is worth noting that this is the only cycle allowed by the proposed reference model. It is well known that supporting arbitrary loops in composition definitions would allow the specification of ambiguous situations that are also difficult to comprehend [76].

6.5 SPECIFYING ITERATIONS

The specification of loops in WSTL is done through the *foreach* task. The foreach task is an extension of the composite task and realizes do-until loops: a set of tasks are iterated until a condition is met. The goal of this task is to enforce a block-oriented specification of loops well known from structured programming.

The foreach task evaluates an expression against data produced by previously executed tasks, generating a value set, and repeats its component tasks once for each value in the set. The execution model of the foreach component tasks follows the same rule as for component tasks of composite tasks. The foreach task terminates once the last component task has completed over the last value in the set.

The foreach task has a named variable and a selection attribute, besides the elements inherited from the composite tasks. The select attribute specifies an XPath expression evaluated against data produced by a previously executed task, resulting in a set of zero or more values. The resulting set is guaranteed to have some internal order, and its values will be accessed by that order. XPath extension functions must be used to guarantee distinct values or a particular order. Each value in the set is bound to the named variable. The bound variable has no namespace associated with it, and is referenced from an XPath expression using the prefix `$`. The foreach task will repeat all its component tasks according to the specified control flow in the same manner as for the composite task. Each execution sequence will be repeated once for each value in the set.

Figure 6.21 shows an example of a foreach task, which processes each item in a purchase order and binds just that one item to the named variable `oneItem`. This value can then be consumed by the component tasks of the foreach task.

```
<wstl:foreachTask id="processItens"
                  name="oneItem"
                  selection=" purchaseOrder/inputMsg/param[name='itens']/item">

 ...

</wstl: foreachTask >
```

Figure 6.21 - Example of a foreach data link.

The data produced, after each repetition, by the component tasks of the composite task must be aggregated in the output message of the foreach task. The data aggregation is

defined through regular data links having the component tasks as its source nodes and the foreach task as its target node. At each repetition, the data produced by the component tasks are appended to the data already stored in the output message of the foreach task.

6.6 WSTL ELEMENTS FOR DEFINING COMPOSITIONS

In this section, we describe all WSTL elements used for defining compositions in its details. The XML Schema of these elements are presented in Appendix 9.

6.6.1 Composition Definition Element

The root definition for specifying WSTL compositions is the `compositionDefinition` element. This element is optionally identified by a name and may contain an unbounded set of the following elements: data links, fault data links, and tasks. Figure 6.22 shows the XML Schema fragment of the `compositionDefinition` element.

```
<xsd:element name="compositionDefinition" type="wstl:compositonType"/>
<xsd:complexType name="compositonType">
    <xsd:choice maxOccurs="unbounded">
      <xsd:element ref="wstl:dataLink"/>
      <xsd:element ref="wstl:faultDataLink"/>
      <xsd:group ref="wstl:task"/>
    </xsd:choice>
    <xsd:attribute name="name" type="xsd:ID" use="optional"/>
</xsd:complexType>
```

Figure 6.22 - XML Schema fragment for the `compositionDefinition` element and the `compositionDefinitionType` type.

6.6.2 Data Link Element

Data links are defined by the `dataLink` element, which is named using the `name` attribute. A `dataLink` element defines an assignment occurring between a source task and a target task. Figure 6.23 shows the XML Schema fragment of the `dataLink` element.

```
<xsd:element name="dataLink" type="wstl:dataLinkType"/>
<xsd:complexType name="dataLinkType">
  <xsd:complexContent>
    <xsd:extension base="wsdl:documented">
      <xsd:sequence>
        <xsd:group ref="wstl:grplink" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="wstl:rule" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="name" type="xsd:ID" use="optional"/>
      <xsd:attribute name="source" type="xsd:QName" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<xsd:group name="grplink">
```

```
  <xsd:choice>
    <xsd:element ref="wstl:link"/>
    <xsd:element ref="wstl:foreachLink"/>
    <xsd:element ref="wstl:variableLink"/>
  </xsd:choice>
</xsd:group>
```

Figure 6.23 - XML Schema fragment for the `compositionDefinition` element and the `compositionDefinitionType` type.

A `dataLink` element can have the following child elements: a `rule` and a set of `link`, `foreachLink` and `variableLink` elements.

The `rule` element is used to evaluate a rule. The rule must evaluate to true for a data link to be consumed. This element is described in Section 6.6.3.

The `link`, `foreach` and `variable` elements are used to perform data assignment between a source and target context. The source task of a data link defines the *source context* and its target task defines a *target context*. The target context depends on the location in which a `dataLink` element is used. The source context is defined by the value of the attribute `source` of the selected data link. The source context and target context are equivalent to the definition given by the XPath specification [140] These elements are described from Section 6.6.2.1 to Section 6.6.2.3.

6.6.2.1 Link Element

A `link` element defines a data assignment occurring between a message part of a source task and a message part of a target task. Figure 6.24 shows the XML Schema fragment of the `link` element.

```
<xsd:element name="link" type="wstl:linkType"/>
<xsd:complexType name="linkType">
  <xsd:complexContent>
    <xsd:extension base="wsdl:documented">
      <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:group ref="wstl:grplink"/>
      </xsd:sequence>
      <xsd:attribute name="append" type="xsd:boolean" use="optional" default="true"/>
      <xsd:attribute name="target" type="wstl:XPathType" use="optional"/>
      <xsd:attribute name="selection" type="wstl:XPathType" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
```

```
<xsd:group name="grplink">
  <xsd:choice>
    <xsd:element ref="wstl:link"/>
    <xsd:element ref="wstl:foreachLink"/>
    <xsd:element ref="wstl:variableLink"/>
  </xsd:choice>
</xsd:group>
```

Figure 6.24 - XML Schema fragment for the `link` element and the `linkType` type.

The `target` attribute specifies a new target context within an existing target context. When this attribute is used, a new target context is created, data is assigned to the existing target context, and all data assignments are performed to the new target context.

A simple data assignment evaluates the XPath expression against the source context and assigns the result to an existing target context, or if the target attribute is used, to a new target context. A complex data assignment defines a target context and performs a series of nested assignments to that target context. The target attribute takes the form of an element or an attribute name. An attribute name is prefixed with `@`.

The `selection` attribute is an XPath expression that operates on the source context and returns an XPath result.

The `append` attribute is a Boolean value. If the `append` attribute is true, the `link` element will append the new elements produced by the `selection` attribute to any existing list of elements in the target context. If the append attribute is false (the default), the `link` element will replace any existing list of elements in the target context. This attribute is most useful when the `link` element is used inside a `foreach` element.

6.6.2.2 ForeachLink Element

The `foreach` element evaluates the XPath expression into a set of zero or more values, and iterates the nested data assignments over that set of values. Each value in the set is bound to the named variable, and the nested data assignments are repeated with the same source and target context, and the new variable bindings. Figure 6.25 shows the XML Schema fragment of the `foreach` element.

```
<xsd:element name="foreachLink" type="wstl:foreachLinkType"/>
<xsd:complexType name="foreachLinkType">
  <xsd:sequence maxOccurs="unbounded">
    <xsd:group ref="wstl:grplink"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:NCName" use="required"/>
  <xsd:attribute name="select" type="wstl:XPathType" use="required"/>
```

```
</xsd:complexType>
```

Figure 6.25 - XML Schema fragment for the `foreachLink` element and the `foreachLinkType` type.

The `select` attribute defines the XPath expression that generates the value set. The `name` attribute identifies the bound variable. The nested data assignments are defined by `link` or `variableLink` elements specified as child elements of the selected `foreachLink` element. The bound variable has no namespace associated with it, and is referenced from an XPath expression using the prefix `$` and the same name as specified by the name attribute.

6.6.2.3 VariableLink Element

The `variableLink` element assigns the result of a simple or complex data assignment into a bound variable. The bound variable is in the global scope of the source task of the selected data link. Figure 6.26 shows the XML Schema fragment of the `foreach` element.

```
<xsd:element name="variableLink" type="wstl:variableLinkType"/>
<xsd:complexType name="variableLinkType">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
    <xsd:group ref="wstl:grplink"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:NCName" use="required"/>
  <xsd:attribute name="source" type="xsd:QName" use="optional"/>
</xsd:complexType>
```

Figure 6.26 - XML Schema fragment for the `variableLink` element and the `variableLinkType` type.

The `source` attribute defines a context that locally changes the source context of the selected data link. The `variableLink` element creates a new target context and performs all simple and complex data assignments to that target context. The target context is bound to the named variable, defined by the attribute `name`, without affecting any existing target context. A bound variable has no namespace associated with it, and is referenced from an XPath expression using the prefix `$` and the same name as specified by the name attribute.

6.6.3 Rule Element

The `rule` element is used to evaluate a rule. The `rule` element uses the `condition` attribute to specify an XPath expression that evaluates to a Boolean value, as in Figure 6.27.

```
<xsd:element name="rule">
  <xsd:complexType>
    <xsd:attribute name="condition" type="wstl:XPathType" use="required"/>
  </xsd:complexType>
</xsd:element>
```

Figure 6.27 - XML Schema fragment for the `rule` element.

A rule is a condition that can be used in the data link, or in a execution dependency element. When referenced by the data link element, rules are evaluated against the contents of a message or the execution state of the source task of the data link. The rule must evaluate to true for a data link to be consumed.

When referenced by the execution dependency element, rules are evaluated against all data consumed by the data links of the task that defines the selected execution dependency. The evaluation of a rule constrains which tasks are candidates for execution.

6.6.4 Atomic task Element

Figure 6.28 shows the XML Schema fragment of the `atomicTask` element that defines atomic tasks.

```
<xsd:element name="atomicTask" type="wstl:atomicTaskType"/>
<xsd:complexType name="atomicTaskType">
  <xsd:complexContent>
    <xsd:extension base="wstl:baseTaskType">
      <xsd:sequence>
        <xsd:element ref="wstl:dataLinkRef" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="wstl:dataLink" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="medService" type="xsd:QName" use="required"/>
      <xsd:attribute name="operation" type="xsd:NCName" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<xsd:complexType name="baseTaskType">
  <xsd:complexContent>
    <xsd:extension base="wsdl:documented">
      <xsd:sequence>
        <xsd:element ref="wstl:dependency" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 6.28 - XML Schema fragment of the `atomicTask` element.

The simpleType element, as well as all other elements representing tasks, is derived from the baseTaskType type. This type has an attribute named id, which is used for identifying tasks and a set of: dependency elements used for defining the execution dependencies having the selected task as its target task; dataLink and dataLinkRef elements used for defining the data links having the selected task as its target task. The dependency element is described in Section 6.6.5. The dataLink element is described in Section 6.6.2.

Besides its derived definitions, the simpleType element has an unbounded set of: dataLink and dataLinkRef elements; and two attributes: medService, and operation.

The dataLink and dataLinkRef elements defines the data links used to produce the input of the atomic task. The context of these data links is the atomic task itself and they are evaluated when the atomic task is scheduled to run.

The medService attribute references a mediator service using a Qname. The operation attribute references an operation of the mediator service identified by the medService attribute. This attribute identifies the operation that implements the selected atomic task.

6.6.5 Dependency Element

Execution dependencies are defined by the dependency element. Figure 6.29 shows the XML Schema fragment of this element.

```
<xsd:element name="dependency" type="wstl:dependencyType"/>
<xsd:complexType name="dependencyType">
  <xsd:complexContent>
    <xsd:extension base="wsdl:documented">
      <xsd:sequence>
        <xsd:element ref="wstl:rule" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="type" type="wstl:executionStateType" use="required"/>
      <xsd:attribute name="source" type="xsd:QName" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
</xsd:complexType>
```

Figure 6.29 - XML Schema fragment of the dependency element.

A dependency element is referenced by a task and defines a control link. The task referencing a dependency element defines the target task of the control link. The

`source` attribute defines the source task of the control link. The `type` attribute selects a state transition condition to be used when evaluating the `dependency` element.

A `Dependency` element can have one `rule` element associated to it. The `rule` element defines a rule that must be evaluated before the command of the state transition dependency of the execution dependency is executed. If the rule evaluates to false then the command will not be performed thus the task will stay in its current state and the dependency will be deactivated.

6.6.6 Composite task Element

Figure 6.30 shows the XML Schema fragment of the `compositeTask` element that defines composite tasks.

```
<xsd:complexType name="compositeTaskType">
  <xsd:complexContent>
    <xsd:extension base="wstl:baseTaskType">
      <xsd:sequence>
        <xsd:element ref="wstl:messages"/>
        <xsd:element ref="wstl:taskRef" maxOccurs="unbounded"/>
        <xsd:element ref="wstl:mandatoryTask"/>
        <xsd:element ref="wstl:dataLink" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="wstl:faultDataLink" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="wstl:dataLinkRef" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="wstl:faultLinkRef" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 6.30 - XML Schema fragment of the `compositeTask` element.

The `compositeTask` element is derived from the `baseTaskType` type already described in Section 6.6.4. Besides the components of its derived type, the `compositeTask` element has a set of six other elements: `messages`, `taskRef`, `dataLink`, `faultLink`, `dataLinkRef`, and `faultLinkRef`.

The `messages` element defines the signature of the composite task. This element is a *request-response* operation (Section 5.1) thus defining the input, output, and fault messages of a composite task.

A `compositeTask` element can have an unbounded set of `taskRef` elements. Each one of these elements references a task and indicates that this task is a component task of the selected composite task. A `taskRef` element can reference any kind of task (atomic, composite, synchronization, and foreach tasks).

The mandatory element contains a set of taskRef sub-elements that references the tasks (among the component tasks) that are mandatory.

A compositeTask element can have an unbounded set of: dataLink, faultLink, dataLinkRef, and faultLinkRef elements. These elements define data or fault links used to produce the output of the composite task. These data and fault links are not in the context of any explicitly defined execution dependencies. The context of these links is the composite task itself and they are evaluated when the composite task terminates its execution.

6.6.7 Synchronization Task Element

Figure 6.31 shows the XML Schema fragment of the synchTask element that defines synchronization tasks.

```
<xsd:element name="syncTask" type="wstl:syncTaskType"/>
<xsd:complexType name="syncTaskType">
  <xsd:complexContent>
    <xsd:extension base="wstl:baseTaskType">
      <xsd:sequence>
        <xsd:group ref="wstl:notification" minOccurs="0"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 6.31 - XML Schema fragment of the compositeTask element.

The syncTask element is derived from the baseTaskType type already described in Section 6.6.4. Besides the components of its derived type, the syncTask element has an optional child element for defining the signature of the synchronization task. This element is a *notification* operation (Section 5.1) thus, it defines only output, and fault messages. The output and fault messages of a syncTask element defines a container for data produced by the synchronized tasks of the selected synchronization task.

6.6.8 Foreach Task Element

Figure 6.32 shows the XML Schema fragment of the foreachTask element that defines foreach tasks.

```
<xsd:element name="foreachTask" type="wstl:foreachTaskType"/>
<xsd:complexType name="foreachTaskType">
  <xsd:complexContent>
    <xsd:extension base="wstl:compositeTaskType">
      <xsd:attribute name="name" type="xsd:string" use="required"/>
      <xsd:attribute name="selection" type="wstl:XPathType" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
```

```
</xsd:complexType>
```

Figure 6.32 - XML Schema fragment of the `compositeTask` element.

The `foreachTask` element is derived from the `baseTaskType` type already described in Section 6.6.4. Besides the components of its derived type, the `foreachTask` element has attributes: `name` and `selection`.

The foreach element is a composite task and extends the type `compositeTaskType`. It can be used in any place where a composite task can appear. The foreach task evaluates an expression against the composition data and repeats its component tasks once for each value in the set. This task completes successfully once the last composite task has completed over the last value in the set.

The `selection` attribute specifies an XPath expression evaluated against the composition data, resulting in a set of zero or more values. The current value can be referenced from an XPath variable identified by the value of the attribute `name` prefixed with `$`.

7. The Execution Model for Transaction Mediation

In this chapter, we describe the model used to coordinate the execution of a given mediator service composition in WebTransact. This execution model has two levels of control. The first is the composite task coordination level and the second is the atomic task coordination level. The first level specifies the rules for interpreting a composite task specification. Such rules ensure that a given composite task will be executed according to the semantics of its specification and guarantee that such execution will run as a single transaction. The second level specifies the rules for coordinating the execution of the mediator service operations that implement atomic tasks. This level is responsible for providing a transactional interface for all atomic tasks, which is used by the first level, in order to coordinate composite task executions. The coordination of atomic tasks deals with problems that arise due to the one-to-many relationship that exists between mediator service operations - that implement atomic tasks - and remote service operations - that are the concrete service providers, as well as with the problems that arise due to the dissimilar transaction behavior of semantically equivalent remote service operations.

The development of the transaction model described in this chapter was guided by the transactional requirements presented in Section 2.1. Therefore, such transaction model exploits the features of the Web service environment, such as the dissimilar capabilities of Web services and the existence of multiple semantically equivalent Web services, and it ensures a relaxed notion of failure atomicity, named *2L-guaranteed-termination* (two-level-guaranteed-termination). The *2L-guaranteed-termination* property is an extension of the *guaranteed-termination* property presented in [5]. The latter property is not general enough to be used as the correctness criterion of the transaction model of WebTransact. The semantic power of the *guaranteed-termination* property is sufficient as the correctness criterion in the coordination of composite task executions. However, it does not contemplate the second level of transaction coordination that exists in the transaction model of WebTransact. Therefore, we have extended the guaranteed-termination property to encompass both the composite task level and the atomic task level, through the 2L-guaranteed-termination property.

7.1 PRELIMINARY CONCEPTS - COMPOSITE TASK MODEL

In this section, we present some basic concepts used to explain the reference model for executing a composite task.

7.1.1 Specifications, Instances and Executions

A composite task specification is the abstract definition of a given composition. It models the tasks as well as its associated wiring by means of control links, data links, and associated rules. An instance of a composite task is its abstract specification ready for execution. Basically, whenever a composite task is instantiated, its input data are materialized and all its component tasks are created. After its instantiation, a composite task can be executed. An execution of a composite task instance consists of determining its start tasks and performing them, receiving output of completed tasks, determining their actual successors, materializing their input, performing them, and so on. An execution actually results in assigning states to the tasks, managing the context of the instance, and invoking, for each atomic task, the mediator service operation that implements it.

The execution of composite task instances is managed by the *mediator task scheduler*. This component is responsible for creating and invoking all component tasks, and enforcing all inter-task dependencies of a composite task, according to its specification, when executing it.

7.1.2 Transaction Behavior of Tasks

The transaction behavior of atomic tasks is based on the transaction behavior of the mediator service operation that implements it[9]. For instance, if a mediator service operation O_i is pivot and O_i implements the task t_i then t_i is also pivot. The transaction behavior of composite tasks is based on the transaction behavior of all its component tasks. If all its component tasks support the compensable or the retriable transaction behavior then the composite task is, respectively, compensable or retriable. Otherwise, if at least one component task supports only the pivot transaction behavior then the composite task is pivot. As an example, suppose that task t_i is a composite task whose component tasks are tasks t_j and t_k. Assume that task t_i exposes the compensable transaction behavior. This means that task t_i can be compensated after its execution. Since

[9] Therefore, the *transaction behavior* of a task can be either *compensable*, *retriable*, or *pivot*.

task t_i is a composite task, its compensation action is the compensation of all its executed component tasks. Therefore, the compensation of t_i leads to the compensation of t_j and t_k, hence t_i is compensable only if t_j and t_k are compensable. A similar reasoning can be applied to infer the other transaction behaviors.

7.1.3 Transaction Lifecycle Interface

Each component task of a composite task is always associated with a port type that allows managing the transactional lifecycle of the task. This port type provides operations for instantiating, terminating, and for controlling the state transitions of a given task. The transactional lifecycle interface is used by the mediator task scheduler to enforce the inter-task dependencies of a composite task when executing it. When the mediator task scheduler receives an instance of a composite task to be executed, it uses the operations defined in the lifecycle interface to create and control the scheduling of all component tasks of that composite task.

The following transaction lifecycle operations are defined in WSTL:

- **Create**, creates an instance of the task. As a result, the `id` of the created task instance is returned.

- **Submit**, starts the execution of a task instance. The task instance is executed using the current values of the input message of the task. The operation returns as soon as the task is started. If the task is an atomic task, the operation invokes the mediator service that implements the task. If the task is a composite task, a new instance of a task scheduler is instantiated to control the execution of the task.

- **Abort**, aborts the execution of a task instance. As a result, if the task instance is in the *executing* state, an `abort_ack` message is returned. If the task instance is in any state other than *executing*, an `error` message is returned.

- **Compensate,** compensate a committed task instance. As a result, if the task instance is in the *committed* state, a `compensate_ack` message is returned. If the task instance is in any state other than *committed*, an `error` message is returned.

- **Terminate**, terminates the task instance. This operation informs the task instance to release all system resources being used and to perform whichever pending operations.

- **GetTransactionBehavior**, queries the task instance regarding its supported transaction behavior. As a result, the transaction behavior of the task is returned.

Not all tasks instances expose all lifecycle operations. Pivot task instances do not expose neither the compensate operation nor the abort operation. Retriable and compensable task instances expose all operations. When a task instance receives an invocation on an unsupported operation, it returns a fault message.

7.1.4 Task Lifecycle

When a task instance receives a *create* message, through its lifecycle interface, it is instantiated and assumes its initial state of *inactive*. For composite tasks, the receiving of a *create* message dispatch other calls to the *create* message of each one of its component tasks. This procedure continues recursively until only atomic tasks have been reached.

A task instance remains in the inactive state until its execution path is activated. At this point, a task instance must change its state from *inactive* to *enabled*. The active execution path is the path that is being currently tried by the mediator task scheduler in a given moment. Since a composite task can have many different execution paths and they must be tried according to its specification, only one execution path is activated per time. An *enabled* task can be reached by the control flow of the composite task. Once a task instance is reached by the control flow, there are two options as follows:

- The task scheduler decides, based on the evaluation of that task execution dependencies, that its instance must be executed. If the task is implemented by a solicit-response or notification operation, it is immediately started. If the task is implemented by a request-response or one-way operation first, its input message is built according to the associated data links and, after that, it is started. Once started, its state will be *executing*.

- The task scheduler verifies, based on the execution conditions, that the task will never be executed, and therefore puts it in the state *discarded*. *Dead-path elimination* will take place for all paths leaving that task (Section 7.4.2).

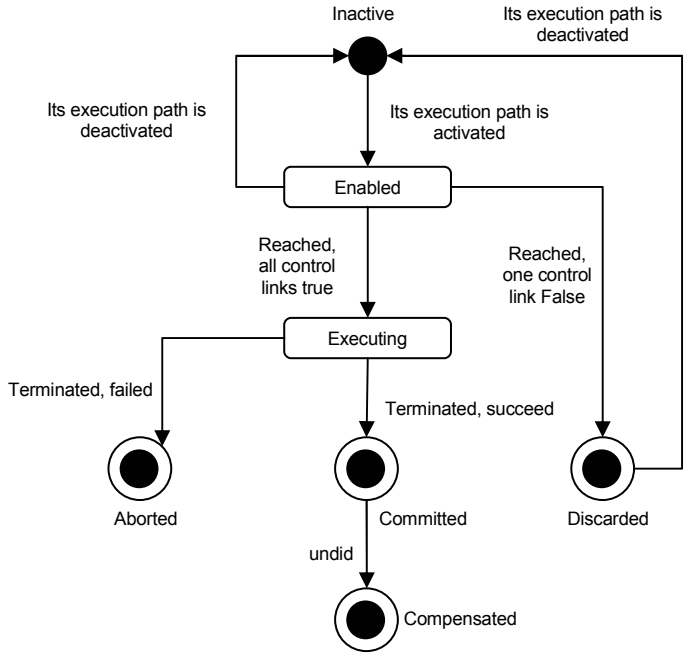


Figure 7.1 - State Transition of Task Instances.

For atomic tasks, after the associated mediator service operation has terminated, the task instance will be in one of the following states: *committed*, *aborted*[10]. In this case, the state transitions to these states depend on the mediator service operation only and are out of the control of the task scheduler. For composite tasks, the states *committed* or *aborted* are reached after all its component tasks has terminated, or an unrecoverable fail has occurred. Due to recovery issues, the task scheduler can decide to undo an already executed task instance. In this case, the task scheduler sends a compensate command to that task instance. Once compensated, the task state is *compensated*.

The execution of the active path can fail if one of its task instances aborts. In this case, if such task is mandatory in that execution path then the task scheduler will try to switch the execution to another execution path. If it succeeds then the active execution path is deactivated. Once an execution path is deactivated, all its task instances in the states *enable* or *discarded* are put in the state *inactive*.

[10] We are not considering network failures. In such case, one more state is necessary, the *unknown state*, where the mediator system has no knowledge on the actual state of a given remote service execution due to the occurrence of network failures (the time-out is reached without answer from the remote service).

7.2 FORMAL DEFINITIONS

In this section, we present some formal definitions for reasoning on the correct execution of composite tasks. In the following, we will consider tasks executed by a transactional task scheduler according to the rules defined in a composite task specification. The task scheduler interacts with tasks through the transactional lifecycle interface provided by each component task. This interface has a limited set of transactional operations as described in Section 7.1.2.

Let $\hat{T}$ be the set of all possible tasks instances. For each invocation of a task of $\hat{T}$, return values are provided. Tasks are atomic by definition and therefore terminate either committing or aborting. Tasks differ in terms of their transaction behavior, they are either *compensable*, *retriable*, or *pivot* (Section 7.1.2). In the case of compensable tasks, a compensation operation is provided by the mediator service that implements the task, retriable tasks are guaranteed to terminate successfully after a finite number of invocations, and pivot tasks are those, which are neither compensable nor retriable. These different termination guarantees of tasks will be formally defined, as follows, using the notion of task sequence to denote the ordered execution of tasks.

Definition 1 (Retriable Tasks)

A task t_i *is retriable if some* $m \in \mathbb{N}$ *exists with* t_{i_j} *terminating with abort for* $1 \leq j < m$ *while* t_{i_m} *is guaranteed to terminate with commit.*

Definition 2 (Effect-free tasks)

Let $\lambda = \langle t_i, t_j \ldots t_n \rangle$ *be a sequence of tasks from* $\hat{T}$*. The sequence λ is effect-free if, for all possible task sequences α and β from* $\hat{T}$*, the return values of α and β in the concatenated task sequence* $< \alpha\, \lambda\, \beta >$ *are the same as in the task sequence* $< \alpha\, \beta >$.

A special case of effect-free activities is the sequence $\lambda = \langle t_i, t_{-i} \rangle$ consisting of a compensable task t_i and its compensating task t_{-i}. **Definition 2** formalizes this case.

Definition 3 (Compensable tasks)

A task $t_i \in \hat{T}$ *is compensable if a retriable task* $t_{-i} \in \hat{T}$ *exists where the task sequence* $\lambda = \langle t_i, t_{-i} \rangle$ *is effect-free. The task* t_{-i} *is called the compensating task of* t_i.

Definition 4 (Pivot Tasks)

*A task t_i is pivot if it is neither retriable (**Definition 1**) nor compensable (**Definition 3**).*

Since there is always one invocation that will commit, retriable tasks have guaranteed its successful termination.

Definition 5 (Task Failure)

A task t_i has successfully terminate if there exists a $m \in \mathbb{N}$ such that t_{i_m} is guaranteed to terminate with commit. Otherwise, the task t_i has failed.

To guarantee the property of compensability, a compensating task t_{-i} is itself not compensable, however, it is retriable and thus guaranteed to commit. Furthermore, according to the flex transaction model both pivot and retriable tasks do not have a compensating task.

The control flow of composite task instances is specified by execution dependencies. These execution dependencies support two different types of control flow: (*i*) execution ordering between two component tasks, and (*ii*) preference ordering between two sets of component tasks. These types of control flows are formally defined by the notion of *precedence order* and *preference order* [151].

Definition 6 (Precedence Order)

*Let $\Psi = \{t_i, t_j \ldots t_n\}$ be a set of tasks from $\hat{T}$. The **precedence order** $\prec$ is a partial order over Ψ such that, if $t_i \prec t_j$ $(i \neq j)$ then t_i precedes t_j.*

The precedence order $\prec$, has temporal semantics. For any two tasks, t_i and t_j, if $t_i \prec t_j$, then t_j can only be executed *after* t_i has committed. Therefore, if t_j has executed in an instant T_2 then t_i must have executed in some instant T_1 such that $T_1 < T_2$.

The preference ordering identifies a priority order of execution among possible *execution paths* of a given composite task execution. Consider a composite task specification *CT*. A control link $t_j \rightarrow t_k$ representing an *abort-start* dependency defines a contingency execution path in *CT*, where the execution control of a given instance CT_i of *CT* can take a contingency path through t_k (the target task), when t_j (the

source task) fails. Due to the existence of *abort-start* dependencies, the execution control flow of *CT* can flow from one task t_i toward conjunctions of disjunctions of other tasks, i.e., either one group of tasks, or another group, or yet another group, and so on. Therefore, if t_i is a task of *CT*, then let O_i denote set of tasks $\langle t_m \ldots t_n \rangle$ in *CT* that are successfully executed in some instance CT_i of *CT,* right after t_i (and $t_i \prec t_m, \ldots, t_i \prec t_n$). If O_i is nonempty, then, its tasks can be partitioned into nonempty sets $\delta_i^1, \ldots, \delta_i^n$ of tasks characterizing the possible flow of control of CT_i after the execution of t_i. In order to define the possible execution flows of an instance CT_i of *CT*, we define a special type of subgraph, called *c-subgraph* [9]. A subgraph *H* is a c-subgraph if and only if, for every transaction t_i of CT_i for which O_i is nonempty, the edges that in *H* are directed away from t_i lead to all nodes of exactly one $\delta_i^1, \ldots, \delta_i^n$.

Definition 7 (Preference Order)

Let $\Psi = \{t_i, t_j \ldots t_n\}$ *be a set of tasks from* $\hat{T}$. *Let* $G = \langle \Psi, \prec \rangle$ *be a graph then:*

- *A rooted subgraph* $G^{t_i} = \langle V^{t_i}, E^{t_i} \rangle$, *with root* t_i *is a maximal subgraph of* G *where:*
  - *A transaction* $t_j \in V^{t_i}$ *if* t_j *is reachable from* t_i; *and*
  - $\langle t_j, t_k \rangle \in E^{t_i}$ *if* $t_j, t_k \in V^{t_i}$ *and* $t_j \prec t_k$.
- *For each* $t_i \in \Psi$ *let* G^{t_i} *be a rooted subgraph and* S^{t_i} *the set of all c-subgraphs of* G^{t_i}. *We define* ***a preference relation*** $\lhd^{t_i}$ *as a partial order over* S^{t_i} *and* $\lhd = \bigcup_{t_i} \lhd^{t_i}$.

The preference order $\lhd$, has priority semantics. For any $\langle S_j^{t_i}, S_k^{t_i} \rangle \in G$, if $S_j^{t_i} \lhd S_k^{t_i}$, then the execution of $S_j^{t_i}$ has a higher priority over the execution of $S_k^{t_i}$. We can also say that $S_k^{t_i}$ is an alternative to $S_j^{t_i}$. More precisely, the preference order

$\lhd$ is defined over pairs of subgraphs of G that share some task t_i and establishes the order in which these subgraphs will be evaluated. Consider the execution graph G of an instance CT_i of a composite task CT, which has two c-subgraphs $S_j^{t_i}$ and $S_k^{t_i}$ such that $S_j^{t_i} \lhd S_k^{t_i}$. Suppose that exist a set of tasks $\langle t_j, t_k \rangle \in G$ such that $t_j \in S_j^{t_i}$ and $t_k \in S_k^{t_i}$. If there are two order constraints $t_i \prec t_j$ and $t_i \prec t_k$ then, if t_k is executed, either t_i must have failed or both t_j and t_{-j} must have been executed. In addition, all tasks succeeding t_j in $S_j^{t_i}$ must have been compensated before t_k can be executed.

Intuitively, a composite task execution is an arbitrary collection of tasks instances.

Definition 8 (Composite task)

A composite task CT is a triple $(T, \prec, \lhd)$ *where* $T \subseteq \hat{T}$ *is a set of tasks,* $\prec$ *is the precedence relation, and* $\lhd$ *is the preference relation as stated in* ***Definitions 6*** *and 7, respectively.*

Therefore, a composite task CT is a set of related tasks on which the precedence and the preference relation are defined. Note that both orders $\prec$ and $\lhd$ are irreflexive, transitive, and non-symmetric.

The combination of the precedence and the preference relations may define many different execution paths of a given composite task. An execution path represents a complete composite task execution.

Definition 9 (Execution Path)

Let t_s *be a start task and* G^{t_s} *a rooted subgraph and* S^{t_s} *the set of all c-subgraph of* G^{t_s}. *Then, the members of* S^{t_s} *are called execution paths*[11] $\prec$ $-eps$ *of* G^{t_s}.

A composite task may have many execution paths, each one representing one valid alternative (contingency) for the composite task execution. The order in which these alternatives are scheduled is defined by the preference order $\lhd$.

[11] It is important to note that we are averloading the word *path, s*ince an execution path is not a linear structure but a subgraph.

Intuitively, if two execution paths $\prec_i -eps$ and $\prec_j -eps$ of composite task *CT* share some prefix and the tasks $\{t_i, \ldots, t_k\}$ immediately following that prefix in the execution where $\prec_i -eps$ fail, then $\prec_j -eps$ can continue the execution of the composite task *CT* from the point where the shared prefix completed. In this case, the set $\{t_i, \ldots, t_k\}$ defines a *switching set* [151] in *CT*. A *switching point* (t_l) is a task in a *switching set*, which relates one $\prec_i -ep$ to another $\prec_j -ep$. The notion of switching sets and switching points are used to define the algorithm for seeking alternative execution paths that are executed when a mandatory task of a given $\prec -ep$ aborts.

Definition 10 (Switching Point)

Let $\alpha, \beta \in S^{t_s}$ *and consider a task* t_i *that is in* α *and* β. *If* O_i *of* t_i *is nonempty and its tasks can be partitioned into nonempty sets* $\delta_i^1, \ldots, \delta_i^n$ *of tasks such that* $n > 1$ *and* $\alpha^{t_i}, \beta^{t_i}$ *are rooted subgraphs, with root* t_i, *of* α *and* β *then* t_i *is a switching point if and only if* $\alpha - \alpha^{t_i} = \beta - \beta^{t_i}$.

From now on, we assume that all composite tasks are labeled with a unique identifier, say CT_i. The component tasks of CT_i are designated as $\{t_{i_1}, t_{i_2}, \ldots, t_{i_n}\}$. The subscript indices define the composite task id and a unique id of the task within the composite task. For example, task t_{i_1} is a component task of composite task CT_i with id *1*. The transaction behavior of a task is identified by superscript indexes. For example, task $t_{i_1}^c$ is a compensable task. The transaction behavior can be omitted when it is not relevant. The commitment of composite task CT_i is denoted by C_i while its abort by A_i.

Figure 7.2 shows three graphs representing different views of the same composite task CT_1. The first graph is the dependency graph and represents the composite task specification[12]. The second graph is the execution graph and represents the execution paths of the composite task as well as the preference relation between those paths. This graph is built using the semantics defined in the specification of the composite task and shows the precedence order, represented by a solid arrow, and the preference order, represented by a dashed arrow. This graph also shows the transaction behavior of each component task. The view consists of a set of graphs representing all possible executions

[12] The definition of composite task specification can be found in Section 7.3.5.

of the composite task. These graphs are derived from the execution graph. Given the orders defined in the execution graph - t_{1_4} and its successor t_{1_5} can only be executed after t_{1_2} or t_{1_3} has failed and t_{1_2} has compensated by t_{-1_2} - four possible executions of CT_1 exist: (*i*) the first one represents the normal execution where only the main path is reached; (*ii*) the second represents the execution where task t_{1_1} fails; (*iii*) the third represents the execution where task t_{1_3} fails; and (*iv*) the last one represents the execution where task t_{1_2} fails. According to the mandatory tasks, only the second execution will lead CT_1 to abort, CT_1 commits in all other executions.

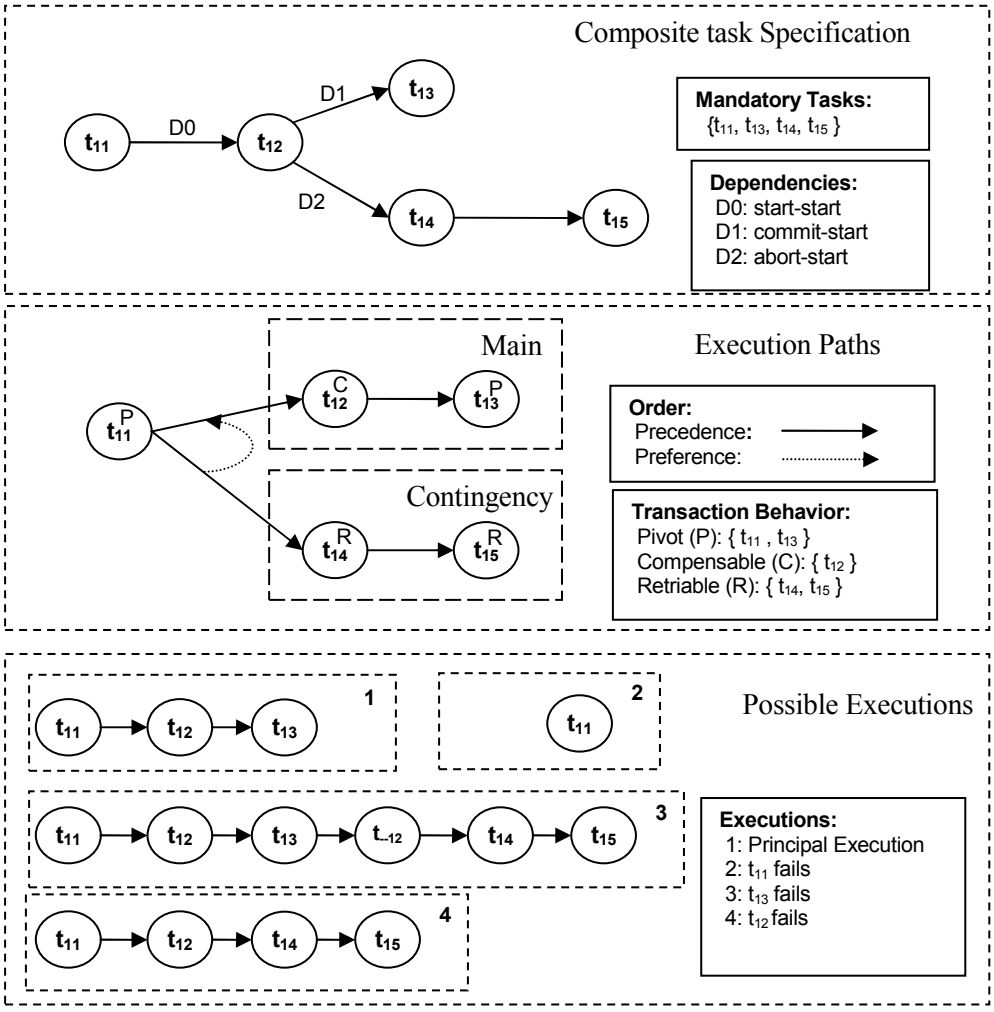


Figure 7.2 -Composite task example, from specification to execution.

Due to the dissimilar transaction behavior of tasks, not all executions of a composite task are *safe*. A composite task execution is safe if, at any time, it is possible to guarantee

that its execution will terminate without leaving any side effects. Suppose that a composite task execution CT_i that has successfully executed task t_{i_i} in one $\prec_i -ep$. If t_{i_i} is a pivot task then, after its execution, the composite task CT_i can no longer be aborted. Intuitively, in this case, only retriable tasks or any kind of non-mandatory tasks can be executed after t_{i_i} has committed. Suppose that another task $t_{i_j} \in \prec_i -ep$ is executed after t_{i_i} in *CT*. If t_{i_j} is, for instance, a compensable and mandatory task[13] in $\prec_i -ep$ then the execution of CT_i, in such $\prec_i -ep$, is *not* safe.

For reasoning on safe executions of a composite task, we introduce the concept of well-formed composite task, which is similar to the concept of *well-formed flex structure* [151] defined for flexible transactions.

In [151] a flexible transaction $\mathcal{T}$ is defined as a set of related subtransactions on which the precedence relation, $\prec$, and the preference relation, $\lhd$, are defined. The definition of composite tasks (**Definition 8**) also establishes the precedence relation, $\prec$, and the preference relation, $\lhd$, but over a set of tasks instead of subtransactions. Since the building blocks of composite tasks are transactional tasks, a composite task execution can be viewed as a flexible transaction. The basic well-formed flex structure consists of a set of compensable or retriable subtransactions, followed by one pivot subtransaction, which is again followed by a set of retriable subtransactions. Additionally, the pivot subtransaction can be recursively succeeded by a complete well-formed flex structure, given that an alternative execution path $\prec_r -ep$, consisting only of retriable subtransaction, exists for it. In [151] it has been shown that well-formed flex transaction structures always guarantee the existence of at least one execution path $\prec -ep$ that can be executed correctly, while all other paths leave no side effects.

The definition of well-formed composite task structure follows the definition of well-formed flex structure, adding the concept of mandatory operation. In the Flex Transaction model, the *critical point* - indicating the boundary line where it is possible to abort a flexible transaction - is defined based on pivot subtransactions. In our mediation model, the critical point is defined in the same way. The difference is in the set of tasks following the critical point. The basic well-formed composite task consists of a set of

[13] If such task is not marked as mandatory, its failure do not influences the composite task execution (definition 11).

compensable or retriable tasks followed by one pivot task followed by a set of other tasks, which are both retriable (mandatory or non-mandatory), or any other kind of non-mandatory task (compensable or pivot).

Definition 11 (Non-mandatory tasks)

Let t_{i_m} *be a task from a composite task* CT_i *and* $\prec_{i_j}$ $-ep$ *an execution path of* CT_i*. A task* $t_{i_m} \in \prec_{i_j}$ $-ep$ *is non-mandatory if a* $k \in \mathbb{N}$ *exists where* $\prec_{i_j}^{k}$ $-ep$ *commits and* $t_{i_m}^{k}$ *fails.*

Definition 12 (Abort-free subgraph)

Let $G^{t_{i_i}}$ *be a rooted subgraph of a composite task* CT_i*, and* $S_j^{t_{i_i}}$ *a c-subgraph in* $G^{t_{i_i}}$*.* $S_j^{t_{i_i}}$ *is abort free if all tasks in* $S_j^{t_{i_i}}$ *are either retriable or any type of non-mandatory task.*

The execution of abort-free tasks will never force a composite task to abort. Retriable task are guaranteed to commit while non-mandatory tasks does not influences the commitment of composite tasks.

Definition 13 (Well-formed Composite task)

A composite task CT_i *is well-formed if, for each* $\prec_{i_j}$ $-ep$ *in* CT_i*, one of the following conditions is satisfied:*

1) There is no task $t_{i_j} \in \prec_{i_j}$ $-ep$ *such that* t_{i_j} *is non-compensable; or*

2) For each task $t_{i_j} \in \prec_{i_j}$ $-ep$ *such that* t_{i_j} *is non-compensable then the following conditions must be satisfied:*

a) If t_{i_j} *is not a switching point of* $\prec_{i_j}$ $-ep$ *then all c-subgraphs* $S^{t_{i_i}}$ *of the subgraph* $G^{t_{i_i}}$ *rooted in* t_{i_j} *are abort free subgraphs.*

b) If t_{i_j} *is a switching point of a switching set SW of* $\prec_{i_j}$ $-ep$ *regarding the contingency execution path* $\prec_{i_k}$ $-ep$ *then:*

i) The set of all successor tasks of t_{i_j} *in* $\prec_{i_j}$ $-ep$ *configure a well formed composite task and the subset* σ_{i_k} *of all successor tasks of* t_{i_j} *in* $\prec_{i_k}$ $-ep$ *configure an abort free set.*

ii) If a successor of a member in SW is not ordered by $\prec_{i_k}$ $-ep$ *with another member in SW, then it is compensable.*

Composite tasks having well-formed structure are called composite tasks with *guaranteed-termination* [5]. The guaranteed-termination property of transactional composite tasks is a generalization of the *all-or-nothing* semantics of traditional ACID transactions, as it ensures that at least one of eventually many valid executions is effected.

The first non-compensable task of a composite task with guaranteed-termination CT_i is called *state-determining* task t_{i_0} of CT_i. All tasks of CT_i preceding t_{i_0} are compensable. Therefore, backward recovery can be performed by successively applying compensation if t_{i_0} fails or if an abort A_i of CT_i is performed before t_{i_0} has committed. Likewise, once t_{i_0} has terminated successfully, forward recovery is guaranteed. From here, a composite task with guaranteed termination can be in any of two states: (i) a composite task CT_i is said to be *forward-recoverable*, $F-REC$, after t_{i_0} has been committed. Otherwise, CT_i is *backward-recoverable*, $B-REC$. The subset of compensating tasks to be executed for recovery purposes of a composite task in state $B-REC$ is its *backward recovery path*. The subset of tasks leading from any task succeeding t_{i_0} to the well-defined termination of a composite task is the *forward recovery path*. The set of tasks of a composite task CT_i to be executed for recovery purposes (either forward or backward) will be called the completion of CT_i denoted by $C(CT_i)$. Note that in the case of CT_i being in state $B-REC$, $C(CT_i)$ consists only of compensating tasks, while, if CT_i is in state $F-REC$, $C(CT_i)$ consists of both compensating tasks (local backward recovery to a state-determining element t_{i_0}), and abort-free tasks. While the failure of one task leads to the execution of the next contingency given by the preference order $\lhd$, the abort A_i of a composite task CT_i in $B-REC$ considers only compensation in backward order and no further contingency

execution paths. Note that the abort A_i of composite tasks in $F - REC$ is not possible, a composite task CT_i in $F - REC$ always commit through the successive trying of its contingency paths. Since there is at least one path that consist of only abort-free tasks, the safe termination of CT_i is guaranteed. The completion $C(CT_i)$ of a composite task CT_i is an important notion for defining the algorithms used to schedule tasks.

7.3 CORRECT SPECIFICATIONS OF COMPOSITE TASK

Before explaining the execution model of composite tasks, it is important to define the notion of correctness of composite tasks specifications. Composite task specifications are built using the elements defined in Chapter 6. Using those elements, a user can define service compositions aggregating a set of mediator service operations and specifying the desired flow of control and data among these operations. In our model, a service composition is specified through composite task specifications, which are represented by the dependency graph as follows (Section 6.1):

Definition 14 (Composite task Dependency Graph)
A composite task specification CTS_i is a directed graph $D(CT_i)$, named dependency graph of CT_i, whose nodes are all tasks T of $\hat{T}$ and whose edges are all $t_{i_m} \rightarrow t_{i_j}$, where $\left(t_{i_m}, t_{i_j} \in T \text{ and } t_{i_m} \neq t_{i_j}\right)$, such that :

1) $t_{i_m} \overset{\Delta}{\underset{\Phi}{\rightarrow}}_{cl} t_{i_j}$ represents a control link [14], where Δ is an execution dependency[15] and Φ is a rule; and

2) $t_{i_m} \underset{\Phi}{\rightarrow}_{dl} t_{i_j}$ represents a data link [16], where Φ is a rule[17].

The above definition provides only a broad picture of the structure of composite tasks specifications. Following only this basic definition, a user can specify incorrect or ambiguous composite tasks specifications. In order to avoid some erroneous situations, it is necessary to define the valid relations between the elements used for specifying composite tasks. In the following sections, we discuss the invalid relations among the

[14] The rule and the execution dependency can be omitted when they are not used in the explanation context.

[15] Execution dependencies are defined in Section 6.1.2.

[16] The rule can be omitted when they are not used in the explanation context.

[17] Rules are defined in Section 6.1.4.

elements that composes such specification in order to seek a more precise delineation of the structure of composite task specifications.

7.3.1 Loops

Composite task specifications are abstract definitions of its executions, i.e., a specification of a given composite task might be instantiated many times, each one generating one concrete composite task execution. Therefore, such specifications must satisfy the definition of composite task execution. **Definition 8** states that both the precedence $\prec$ and the preference $\lhd$ relations that exists in composite task executions are irreflexive, transitive, and non-symmetric. Control links representing start dependencies define the control flow in composite task specification thus, they are responsible for generating the precedence $\prec$ and the preference $\lhd$ relations of a given execution. As a direct result from the above observation, such control links *cannot* introduce loops in composite task specifications. The specification of arbitrary loops would generate symmetric precedence relations that is not allowed by **Definition 8**. Furthermore, supporting arbitrary loops in composite tasks specifications would allow the specifications of ambiguous situations that are also difficult to comprehend [76]. More formally:

Definition 15 (Loop)
Let $D(CT_i)$ be a dependency graph and CT_i an execution of $D(CT_i)$. If there is a control link $t_{i_j} \overset{\Delta}{\rightarrow}_{cl} t_{i_m} \in D(CT_i)$ such that Δ represents a start dependency then the following conditions must be satisfied:

1) $t_{i_j} \prec t_{i_m} \in \prec_i$; *and*

2) $t_{i_m} \prec t_{i_j} \notin \prec_i$.

7.3.2 The Alternative Constructor

Tasks and control links used to specify the alternative constructor must obey two rules. First, all synchronized tasks of an alternative constructor can be the source task of only one control link and that control link must reaches a synchronization task. No other control links can leave such tasks. The other rule is related to control links in the opposite direction regarding the first rule, i.e., those that leave a synchronizing task and reach a synchronized task. Such control links can only represent the *commit-compensate*

dependency. Such control links introduce loops in the composite task dependency graph. However, these controlled loops do not introduce any abnormal order in the precedence $\prec$ and/or the preference $\lhd$ relations of composite task executions. In this case, three different situations can occur depending on the execution state of the target task: (*i*) if it is being executed when the control link is activated then that task will be aborted; or (*ii*) if it has been already terminated and it is in the *committed* state when the control link is activated then the task will be compensated; or (*iii*) if it has been already terminated and it is in the *aborted* state when the control link is activated then it will remain in its current state. It is easy to see that, among the situations described above, only the second influences the relation $\prec$. However, this kind of *specification* loop does not introduce a cycle in the composite task execution simply because the compensation of a task t_i is actually performed by a different task t_{-i}.

Definition 14 (Alternative Tasks)
Let $D(CT_i)$ be a dependency graph of a composite task CT_i. If $t_{i_j} \in D(CT_i)$ and t_{i_j} is a synchronization task then the following conditions must be satisfied:

1) *If there is a control link $t_{i_m} \rightarrow_{cl_i} t_{i_j} \in D(CT_i)$ then there is no control link $t_{i_m} \rightarrow_{cl_i} t_{i_k}$ $(\forall\ k \neq j)$ such that $t_{i_m} \rightarrow_{cl_i} t_{i_k} \in D(CT_i)$; and*
2) *If there is a control link $t_{i_j} \overset{\Delta}{\rightarrow}_{cl_i} t_{i_m} \in D(CT_i)$ then Δ is a commit-compensate dependency and there is another data link $t_{i_m} \overset{\Delta}{\rightarrow}_{cl_i} t_{i_j}$ such that $t_{i_m} \rightarrow_{cl_i} t_{i_j} \in D(CT_i)$ and Δ is a start dependency.*

7.3.3 Data Links

A data link can be specified only if its target task is reachable from its source task through a path of directed control links belonging to the same execution path. The goal of this constraint is to avoid some incorrect specifications regarding data consumption. For example, the specification of data links between target tasks willing to consume data from a source task that has not finished its execution yet, or dead lock situations in which one task requires data from another task as input but the later task needs the output of the former as its input.

Definition 17 (Data Links)
Let $D(CT_i)$ be a dependency graph and CT_i an execution of $D(CT_i)$. If there is a data link $t_{i_j} \rightarrow_{dl_i} t_{i_m} \in D(CT_i)$ then there is a precedence relation $\prec_{i_k} \in CT_i$ such that $\left(t_{i_j}, t_{i_m}\right) \in \prec_{i_k}$ and there is an $\prec_{i_k} - ep$ in CT_i.

7.3.4 Non-mandatory Tasks

The last group of constraints for specifying composite tasks is related to non-mandatory tasks. If a task is marked as non-mandatory, all control links leaving such task, if any, must be labeled with a *commit_abort-start* dependency, and no other dependencies are allowed. In addition, it is not allowed to have data links leaving optional tasks and reaching other tasks. The goal of these constraints is to avoid incorrect specification regarding the transactional semantics of tasks. Since non-mandatory tasks does not influence the final state of its composite task, it is meaningless to define data or control dependencies between optional tasks and other tasks of the composition.

Definition 18 (Non-Mandatory Task Specification)
Let $D(CT_i)$ be a dependency graph of a composite task specification CTS_i. If $t_{i_j} \in D(CT_i)$ and t_{i_j} is a non-mandatory task then the following conditions must be satisfied:

1) *If $t_{i_j} \rightarrow_{cl_i} t_{i_m}$ $(j \neq m)$ is a control link in $D(CT_i)$ then $t_{i_j} \rightarrow_{cl_i} t_{i_m}$ represents a commit_abort-start dependency; and*
2) *There is no $\rightarrow_{dl_i}$ in $D(CT_i)$ such that $t_{i_j} \rightarrow_{dl_i} t_{i_m}$ $(j \neq m) \in D(CT_i)$.*

7.3.5 Composite Task Specification

From Section 7.3.1 to 7.3.4, we have defined all the valid relations between the elements used for specifying composite tasks. Based on such definitions, we formally define the structure of a correct composite task specification (**Definition 19**).

Definition 19 (Composite Task Specification)
A composite task specification CTS_i is represented by a directed graph $D(CT_i)$, named dependency graph of CT_i, whose nodes are all tasks T of $\hat{T}$ and whose edges are all $t_{i_m} \rightarrow t_{i_j}$, where $\left(t_{i_m}, t_{i_j} \in T \text{ and } t_{i_m} \neq t_{i_j}\right)$, such that $t_{i_m} \overset{\Delta}{\Phi}\rightarrow_{cl} t_{i_j}$

represents a control link and $t_{i_m} \ {}_{\Phi}\!\rightarrow_{dl} \ t_{i_j}$ *represents a data link. Each control link* $t_{i_m} \ {}^{\Delta}_{\Phi}\!\rightarrow_{cl} \ t_{i_j} \in D(CT_i)$ *satisfies the conditions stated in* ***Definitions 15****,* ***16****, and* ***18*** *while each data link* $t_{i_m} \ {}_{\Phi}\!\rightarrow_{dl} \ t_{i_j}$ *$D(CT_i)$ satisfies the conditions stated in* ***Definitions 17*** *and* ***18****.*

After having defined the structure of a correct composite task specification, we present **Theorem 1**.

Theorem 1
If an execution of a correct composite task specification CTS_i generates a well-formed composite task, then the guaranteed-termination property is ensured for such execution.

Argument
We first argue that correct composite task specifications can generate well-formed composite task executions. A composite task specification is correct if it follows the rules stated in Definition 19. According to such definition, a correct composite task specification generates only executions where the precedence $\prec$ *and preference* $\lhd$ *relations are irreflexivy, transitive, and non-symmetric thus satisfying the basic requirements for generating a well-formed composite task execution. Now, given an execution CT_i of a correct composite task specification CTS_i, if CT_i is well-formed then, during its execution, at any time CT_i is guaranteed to be forward-recoverable* $F - REC$ *or backward-recoverable* $B - REC$*. Since CT_i is* $F - REC$ *or* $B - REC$ *at any execution point, the guaranteed termination property is ensured for CT_i .□*

In the next section, we describe the execution model of composite tasks. We consider all the specification of composite tasks to be correct as stated in **Definition 19**.

7.4 EXECUTION MODEL OF COMPOSITE TASKS

The execution of a composite task consists of two different phases: the *verification phase* and the *scheduling phase*. These steps are detailed in the next sections.

7.4.1 Verification phase

The goal of the verification phase is to ensure the *guaranteed-termination* property of a given composite task instance. This phase receives a specification of a composite task

and returns, as a result, the corresponding execution graph of the composite tasks, or an error message if the execution of that specification does not ensure the *guaranteed-termination* property of composite tasks.

Since atomic tasks may be implemented by no-compensable mediator service operations, it is necessary to verify whether a composite task instance can be executed without leaving any side effects. In the verification phase, all component tasks are queried about their supported transaction behavior in order to analyze the structure of the resulting task execution. If the resulting structure is a *well-formed composite task* structure then the instance is safe, i.e., it ensures the guaranteed-termination property, and can be scheduled to run. Otherwise, the instance is not safe and the WebTransact system returns an error message to the caller program.

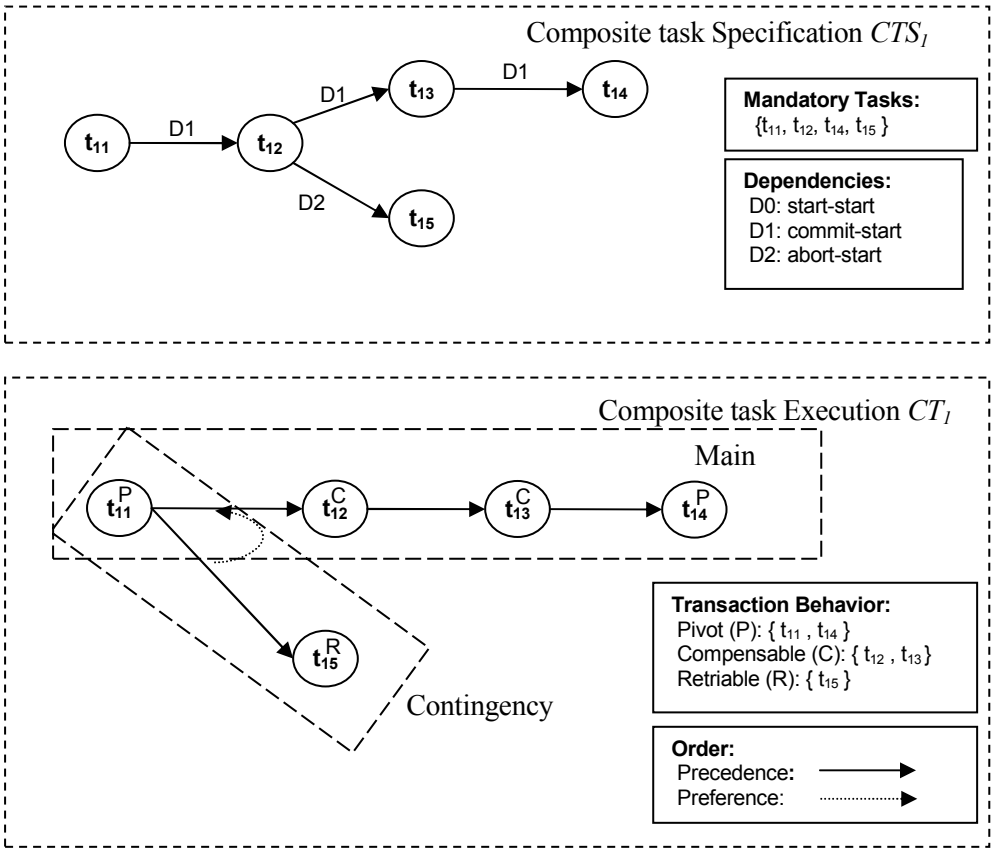


Figure 7.3 - Example of a composite task dependency graph and its corresponding execution graph.

For example, consider the dependency graph CTS_1 showed in Figure 7.3. If CTS_1 must be scheduled to run, the first step of its execution will be the verification phase.

CTS$_1$ will then be used as the input data for the verification algorithm. Using the control links defined in *CTS$_1$*, the verification algorithm builds the execution graph *CT$_1$* and creates an instance of each one of the tasks belonging to *CT$_1$* (sending a `create` message to the life cycle interface). After that, the algorithm sends a `getTransactionBehavior` message to each task instance in *CT$_1$*. After receiving the response of all tasks in *CT$_1$*, the algorithm updates *CT$_1$* with the supported transaction behavior of its component tasks. Then, the resulting execution graph *CT$_1$* is checked regarding its guaranteed-termination property. Returning to Figure 7.3, assume that tasks t_{1_1}, t_{1_4} has returned the value *pivot*, t_{1_2}, t_{1_3} *compensable*, and t_{1_5} *retriable*. The task t_{1_1} is the first non-compensable task of *CT$_1$* thus, it is the *state-determining* task of *CT$_1$*, named t_{1_0}. Clearly, *CT$_1$* is well-formed. The task t_{1_0} of *CT$_1$* is followed by a subset of tasks that is itself well formed-formed and there is a contingency execution path, $\prec_{1_2}$ $-ep$ composed by tasks t_{1_1}, t_{1_5}, rooted in t_{1_0} that is abort-free (t_{1_5} is *retriable*). Therefore, *CT$_1$* ensures the guaranteed-termination property and can be executed. In this example, the verification algorithm return, as a result, an executable instance of the composite task represented by *CT$_1$*.

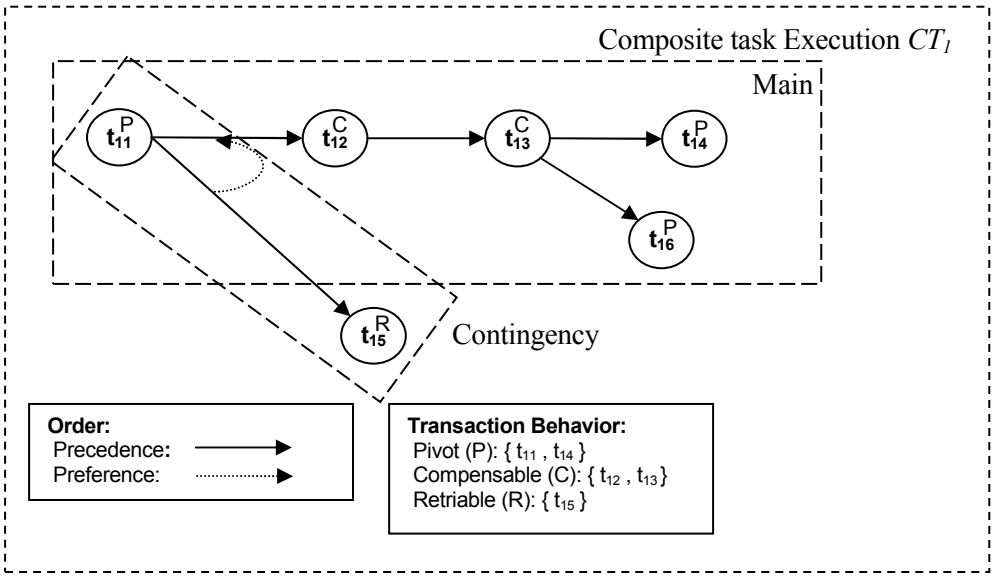


Figure 7.4 - A variation of the composite task example of Figure 7.3.

Now, let us consider a variation of Figure 7.3 in which another pivot task t_{1_6} is the successor of t_{1_3}, as shown in Figure 7.4. In this case, *CT$_1$* is not well-formed. The task

t_{1_0} of CT_1 is followed by a subset of tasks $\Psi_{1_1} = \{t_{1_2}, t_{1_3}, t_{1_4}, t_{1_6}\}$ that is not well-formed. Tasks t_{1_4} and t_{1_6} are both pivot and are not ordered by any precedence relation of CT_1. Therefore, these tasks might be executed in any order, regarding each other. For instance, if t_{1_4} commits and t_{1_6} aborts, it is then impossible for CT_1 either to switch from the main execution path to the contingency execution path or to have the partial effects of the main execution path undone. Since CT_1 does not ensure the guaranteed-termination property, it cannot be executed. In this example, the verification algorithm returns, as a result, an error message indicating that CT_1 is not safe to be executed.

Algorithm 1 describes the verification phase steps:

Algorithm 1 (Verification Phase)
Input: D(CT), a dependency graph of the composite task to be executed.

1. *Build the execution graph CT using the input data.*
2. *Instantiate the composite task instance CT (all component tasks are instantiated).*
3. *For each task instance in CT, send a transactionBehavior message to it.*
4. *After receiving the response from all task instances, check if all pivot tasks in CT satisfies the conditions stated in Definition 13.*
5. *If all tasks have passed the item 4 check, return the composite task instance CT.*
6. *Otherwise, return the error message "The task execution is not safe".*

7.4.2 Scheduling phase

Instances of composite tasks that have passed through the verification phase are the input data of the scheduling phase. In this phase, the task scheduler schedules the component task of a composite task instance according to their execution dependencies.

When the task scheduler receives a task instance to be scheduled, its first step consists of inferring the *mandatory-compensating* constraint of each task instance in $D(CT_i)$. This constraint is a Boolean variable whose value is true when its associated task instance might be compensated after its execution. The *mandatory-compensating* constraint is inferred based on abort-free tasks (**Algorithm 5**). The goal of this constraint is to improve the resource utilization of the overall WebTransact system. Since atomic tasks are implemented by mediator service operations, which may aggregate remote service operations with different transaction behavior, it is important to define, for a given

execution, what kind of transaction behavior is really necessary to be supported. For example, consider a mediator service operation that supports the compensable transaction behavior. Such operation can aggregate remote service operation both compensable and pivot. If this mediator operation is invoked by a task instance associated with a mandatory-compensable constraint whose value is false then it can use all its aggregated remote operations to attend that request. On the other hand, if the value of the mandatory-compensable constraint is true, only those compensable remote operations can be used.

Once the mandatory-compensable constraint is inferred, the task scheduler determines all task instances that have control links whose source tasks are the empty task (these control links represents *start-start* execution dependencies). These task instances define the start tasks of the composite tasks CT_i and also select the *active* execution path, $\prec_{i_j} -ep$, which is the first execution path to be tried.

The task scheduler actually starts the execution of the composite task CT_i when it sends a `submit` message to each one of the start tasks of CT_i. This message is sent along with the value of the mandatory-compensable constraint. Whenever a task instance receives a `submit` message its data links and rules are all evaluated and performed. Based on the start tasks, the *normal navigation* through the dependency graph representing the composite task is started. That means, when a start task completes, its actual successors are determined based on the control links originating at the completed start task. This navigation continues until a task having no control links leaving it completes (called *end* task). The navigation stops at this point because there are no possible follow-on tasks and thus, no actual successor to determine. However, navigation might continue in other parts of the graph, other task instances of the composite task might still be awaiting their execution. However, if all end tasks within the same execution path have been reached, the overall composite task is done. When the last end task of an execution path $\prec -ep$ completes, the output of the overall composite task is determined and returned to its invoker; and then, the composite task execution ceases to exist. In this case, the completion $C(CT_i)$ of the composite task instance CT_i is reached. Note that the completion $C(CT_i)$ does not require that all task instances of $\prec -ep$ have committed. If $\prec -ep$ has executed non-mandatory task instances then it is possible that some of them have aborted. Even though, the overall composite task instance CT_i completes successfully.

The above paragraph described the navigation produced by the scheduling algorithm whenever all mandatory tasks of a $\prec_{-ep}$ have committed. When the task scheduler receives an abort message from any mandatory task instance, it seeks a contingency execution path $\prec_{-ep}$ of the composite task CT_i. Such a contingency execution may be obtained directly, or may be reached by finding a contingency from a more remote predecessors and trying from an early point in the navigation. The task scheduler always seeks for the closest contingency path and it ensures that the preference order $\lhd$ of CT_i is respected. When a contingency path is found, the switching set is determined and all tasks belonging to it are compensated. After that, the navigation is switched to that new $\prec_{-ep}$ and proceeds as the normal navigation. If it is not possible to find a contingency path, the tasks scheduler sends `compensate` messages to all committed tasks instances and `abort` messages to all executing tasks instances. When the last task instance reply with an `aborted` or `compensated` message, the output (fault messages) of the overall composite task is determined and returned to its invoker; and then, the composite task execution ceases to exist. In this case, the completion $A(CT_i)$ of the composite task instance CT_i is reached. Clearly, the above case only happens if the navigation has not reached the state-determining task t_{i_0} of CT_i.

Whenever a task instance t_i completes, all control links leaving t_i are determined and all its execution dependencies are evaluated as well as its associated rules in their actual parameters. The target tasks of all control links whose execution dependencies and rules evaluated to true are exactly the task instances to be scheduled next within the composite task. Some task instances, called *join* tasks, can have more than one control link reaching it. For join tasks, the decision whether it is to be scheduled or not is deferred until all control links finally reach it. However, there is an exception for this rule. If a task instance is reached by more than one control link representing the *weak_commit-start* execution dependency, all these control links are evaluated together. When the first of such set of control links reaches a join task instance (its execution dependencies and rules evaluated to true), all the other control links in that set are considered to have evaluated to true. In this case, if that tasks instance has no other control links reaching it, then it is scheduled to run.

During its navigation throughout the dependency graph, the task scheduler must decide which are the tasks instances that must be scheduled to run. This is done based on

the control links that reaches those tasks. Since tasks instances can participate in more than one execution path, control links from different execution paths may reach it. Consider the fragment of the dependency graph $D(CT_i)$ of Figure 7.5. $D(CT_i)$ has two different execution paths: $(t_B, t_C, t_D) \prec_1$ and $(t_A, t_C, t_D) \prec_2$ such that $\prec_1 \triangleleft \prec_2$. Assume that $\prec_1$ is the active execution path and that task t_B has completed successfully. Therefore, the control link $t_B \xrightarrow{\Delta_1}_{cl} t_C$ will reach task t_C. Since that task has another control link, $t_A \xrightarrow{\Delta_1}_{cl} t_C$, reaching it, its execution should be deferred until it is reached by $t_A \xrightarrow{\Delta_1}_{cl} t_C$. However, this control link will never reach task t_C simply because its source task t_A does not belong to the active execution path, i.e., $\prec_1$. In order to avoid this situation, once the task scheduler activate a given execution path, all its task instances are put in the state *enabled* while all other tasks remains in the state *inactive*. Therefore, the task scheduler considers any control link leaving tasks belonging to the active execution path and discharges the other, i.e., those whose source tasks are in the *inactive* state.

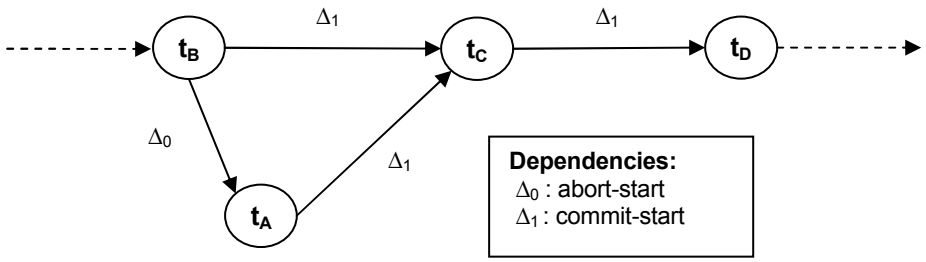


Figure 7.5 - Fragment of a composite task dependency graph with two different execution paths.

Another erroneous situation can occur when navigating through the composite task dependency graph during a composite task execution. Figure 7.6 shows a composite task with start tasks t_A and t_B. Thus, tasks t_A and t_B are scheduled to run when the composite task is instantiated. Assume that task t_A completes successfully and that rule Φ_2 evaluates to true. Also, assume that task t_B completes successfully but the rule Φ_1 evaluates to false.

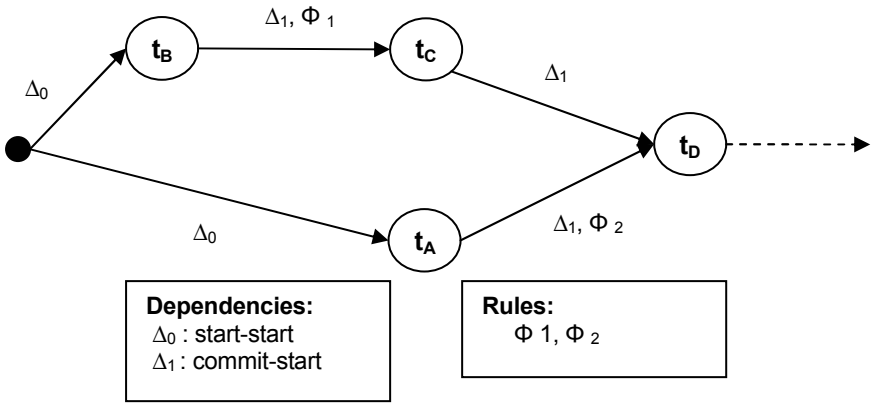


Figure 7.6 - Erroneous behavior of composite tasks execution.

Considering the condition described above, task t_C will never be schedule to run. However, if t_C will not be executed, the control link $t_C \xrightarrow{\Delta_1}_{cl} t_D$ will never be evaluated. Therefore, the task t_D will never be schedule to run. This erroneous situation is avoided through the *dead-path elimination* [76].

Dead-path elimination has to take place whenever it becomes clear that a particular task, of the active execution path $\prec_{ep}$, will never be performed. This is the case, when a rule associated with execution dependency evaluates to false. Originating from that task, dead-path elimination has to traverse the underlying dependency graph of the composite task, throughout the active execution path $\prec_{ep}$, until the next join task or end task is reached.

During this traversal, all visited tasks have to have its state changed from *enabled* to *discarded*. Assume that a join task is reached, the transversal is terminated and the last visited control link is forced to evaluate to true. Therefore, this join task will not be blocked by that control link. However, the decision of its execution will continue to depend on the other active control links reaching it.

Besides the navigation rules described in the above paragraphs, there are special rules when a synchronization task instance is reached. The synchronization task synchronizes the work done by a set of different tasks, its synchronized tasks. The abort and commit of such task is determined by both the termination states of each one of its synchronized tasks and the kind of control links that reaches it. If all control links

reaching a synchronization task represent the *weak-commit-star* dependency then it commits if at least one of these control links evaluates to true (one synchronized task has committed). On the other hand, if all control links represent the *commit_abort-start* dependency then the synchronization task commits if at least one, among its synchronized tasks, has committed. In all other cases, the synchronization task is aborted. After the abort or commit of such task, all control links leaving it and reaching its synchronized tasks are evaluated and its state transition commands performed. At this point, the scheduler algorithm resume to its regular navigation.

7.4.3 Task Scheduler Algorithms

In this section, we present the algorithms used by the task scheduler to interpret composite task specifications. First we show the navigation algorithm responsible for the general execution of a given tasks instance. Then the backtracking algorithm is presented. Next, we present the dead-path-elimination algorithm. Finally, the algorithm for determining the mandatory-abort constraint of task instances is described.

Algorithm 2 (Navigation)
Input: $D(CT_i)$*, the dependency graph of the composite task to be executed.*

Output: $C(CT_i)$*, or* $A(CT_i)$.

Auxiliary Variables: Q*, a queue to store completed tasks.*

1. *When a new instance of composite task is created:*

1.1 Determine all start component tasks in $D(CT_i)$.

1.2 Set the execution path of those tasks as the active execution path $\prec_{ep}$.

1.3 For each task t_i *belonging to* $\prec_{ep}$*, set the state of* t_i *to enable.*

1.4 If the composite task instance is a foreach task:

(a) Compute the value of the foreach bound variable.

(b) Assign the result value of (a) to the input messages of the start tasks according to the semantics of its data links.

1.5 Otherwise:

(b) Assign the input of the composite task to the input of the start tasks according to the semantics of its data links.

2. *Send a submit message to each one of the start tasks.*
3. *From then on, the navigation takes place whenever a task instance returns, that is:*

4. *On receiving response enqueue the response in* Q.

5. *While* Q *is not empty do:*

5.1 Current task instance = dequeue Q

(a) If task instance does not belong to $\prec_{ep}$

(i) Enqueue task instance, resume to (5.1).

5.2 If the task instance succeed or it is a non-mandatory task instance in $\prec_{ep}$ *:*

(a) If the task instance is a synchronization task:

(i) Select all synchronized tasks of that task in the executing state. Send an abort message to these tasks.

(ii) Select all control links leaving that task and reaching its synchronized tasks, such that their state transition conditions evaluated to true.

(iii) For each target task of those control links perform the state transition command.

(b) Select all control links leaving that task and reaching enabled tasks, such that their state transition conditions evaluated to true.

(c) Compute the actual values of the formal parameters of any rule associates with the selected control links and evaluate those rules.

(d) Determine the set of control links that evaluates to true or to false according to the truth-values of these Boolean expressions - item(b) and item(c).

(e) Determine all target tasks of the control links that evaluates to false.

(i) If the result set is one synchronization task and all other control links reaching such task have already been evaluated to false then set the state of this task to aborted and enqueue such task in Q.

(i) Otherwise, perform dead-path-elimination (Algorithm 4) originating at each of these tasks.

(e) Determine all target tasks of the control links that evaluates to true, such that there is no control link reaching them and leaving enabled task instances that have not been evaluated yet.

(i) Compute the input message of each one of the selected tasks based on its data links.

(ii) Compute and apply all data assignments defined for each data link.

5.3 If the task instance fails, perform a backtracking search (Algorithm 3) originating at that task.

5.5 If the backtracking search (5.3) returns `A(CT)` *then resume to (7).*

6. *If there is no task instance* `t` *in* `Q` *such that* `t` *belongs to* $\prec_{-ep}$ *then, resume to (6.1) - there is no more task instances to be scheduled. Otherwise, resume to (5.1).*

6.1 If the composite task instance is a foreach task:

   (a) Compute and apply all data assignments defined for each data link of the composite task instance.

   (b) Compute the next value of the foreach bound variable.

   (c) If there are no more values to be bound, resume to (7).

   (d) Otherwise, assign the result value of (b) to the input messages of the start tasks according to the semantics of its data links. Resume to (2).

6.2 Otherwise, resume to (7).

7. *Terminate the composite task.*

   (a) Compute and apply all data assignments defined for each data link of the composite task instance.

   (b) If `A(CT)` *then return* `A(CT)`*. Otherwise, resume next.*

   (c) Send a terminate message to each one of the task instances of the composite task whose execution states are committed or aborted.

   (d) return `C(CT)`.

Algorithm 3 describes the backtracking algorithm. The backtracking procedure is invoked whenever a mandatory task aborts. The algorithm tries to find the closer contingency execution path to the aborted task. If such path is found, all successors' tasks of the switching task regarding the contingency path are determined. Those tasks that do not participate in any other execution path with greater priority regarding the active path are compensated or aborted, according to their current execution state. Next, the algorithm verifies if the found contingency execution path has any already aborted task, which is mandatory. This situation can occur if the contingency path shares some task with an already executed path. If that occurs, the algorithm will repeatedly try another contingency path. Otherwise, the found contingency path is made active and the algorithm returns.

Algorithm 3 (Backtracking)

Input: $D(CT_i)$*, a dependency graph of the composite task to be executed;* `t`*, the aborted task; and* $\prec_{-ep}$*, the active execution path.*

Output: $D(Ct_i)$*;* t*, the task where the navigation should be resumed;* $\prec_{-ep}$*, the new active execution path; and* $Q(T)$*, a queue of tasks.*

1. *If* t *is a switching point, determine the switching set* SW *that includes* t*. Otherwise, find the closest predecessor* t_1 *in* $\prec_{-ep}$ *that is a switching point, and determine the switching set* SW *that includes* t_1*.*
2. *If a switching set* SW *is found:*

2.1 Determine the set of all tasks in SW *that do not belong to any other execution path* $\prec_{i-ep}$*, such that* $\prec_{-ep} \lhd \prec_{i-ep}$*.*

(a) For each task t_i *of this set:*

(i) if t_i *is in the executing state then send an abort message to it;*

(ii) if t_i *is in the committed state then send a compensate message to it.*

2.2 Determine all tasks in SW *in the executing state and all tasks in the committed state and have not participated in the navigation yet. Insert those tasks in* $Q(T)$*.*

2.3 Set the contingency execution found in (1) path as the new active execution path $\prec_{1-ep}$*.*

2.4 Determine all mandatory tasks in $\prec_{1-ep}$ *that have been already aborted.*

(a) If there is any aborted task:

(i) Select the aborted task with the greater precedence order in $\prec_{1-ep}$*.*

(ii) Set this task instance as t*.*

(iii)Resume to (2).

(a) Otherwise:

(i) Resume to (2.5)

2.5 Set the state of all enabled and discarded tasks instances in SW *as inactive.*

2.6 Set the state of all inactive tasks instances in $\prec_{1-ep}$ *as enable. Return* $D(CT_i)$*,* t_1*,* $\prec_{1-ep}$*, and* $Q(T)$*.*

3. *If no switching set is found, send an abort message to all task instances in* $\prec_{-ep}$ *in the executing state. Send a compensate message to all task instances in* $\prec_{-ep}$ *in the committed state. Return* $A(CT_i)$*.*

Algorithm 4 describes the dead-path-elimination procedure. This algorithm transversal the active execution path of the composite task dependency graph, starting

from a given task t, and set the state of all task instances reached during the transversal to *discarded*. The transversal stops when an end task or a join task is reached.

Algorithm 4 (Dead-Path Elimination)
Input: $D(CT_i)$, *a dependency graph of a composite task;* t, *the blocked task; and* $\prec_{-ep}$, *the active execution path.*

Output: $D(CT_i)$, *the dependency graph received as input.*

1. *Starting from* t, *visit all tasks* t_i *reached by all control links in* $\prec_{-ep}$ *that leaves* t.
2. *If* t_i *is a join task or an end task in* $\prec_{-ep}$, *mark the control link used to visit* t_i *as true.*
3. *Otherwise, set the state of* t_i *to discarded. Recursivelly visit all tasks* t_i *reached by all control links in* $\prec_{-ep}$ *that leaves* t_i. *Resume to step 2.*
3. *When there are no more tasks to visit, return* $D(CT_i)$.

The following algorithm describes the procedure for determining the mandatory-abort constraint of each task instance in a dependency graph of a given composite task. This algorithm transversal the dependency graph seeking task instances followed only by abort-free tasks or that are switching points having contingency paths that configure an abort-free set. The mandatory-abort constraint of such tasks is then set to false.

Algorithm 5 (Mandatory-compensating Constraint)
Input: $D(CT_i)$, *a dependency graph of a composite task.*

Output: $D(CT_i)$, *the dependency graph received as input modified by the mandatory-compensating constraint.*

1. *Set the mandatory-compensating constraint of all task* t_i *in* $D(CT_i)$ *to true*
2. *Determine all task* t_i *such that* t_i *is a switching point of some* $\prec_{-ep}$ *of* $D(CT_i)$.

2.1 For each task t_i *determine all contingency paths* $\prec_{c\ -ep}$ *rooted in* t_i.

(a) *If there is a* $\prec_{c\ -ep}$ *such that* $\prec_{c\ -ep}$ *configures an abort-free set then set the mandatory-compensating constraint of* t_i *to false.*

3. *Determine all execution path* $\prec_{i\ -ep}$ *of* $D(CT_i)$.

2.1 For each $\prec_{i\ -ep}$ *:*

(a) *Determine all subsets* Ψ_i *of* $\prec_{i\ -ep}$ *such that* Ψ_i *configures an abort-free set and contains an end task of* $\prec_{i\ -ep}$.

(b)If there is no tasks t_i, t_j *belonging to* $\prec_i - ep$ *such that* t_i, t_j *is not ordered by* $\prec_i - ep$ *and* t_i, t_j *do not belong to any subset* Ψ_i *then:*

(i) For each subset Ψ_i*, set the mandatory-compensating constraint of all its tasks to false.*

3. Return $D(CT_i)$.

7.5 EXECUTION MODEL OF ATOMIC TASKS

The execution model of atomic tasks defines how an invocation of a given abstract mediator service operation is, in fact, realized by concrete remote service providers. Atomic task instances are submitted for execution during the scheduling phase of composite tasks. Such event triggers the invocation of the mediator service operation that implements that atomic task. Since the logic of a mediator service operation is implemented by its aggregated remote service operations, when such operation is invoked, it is necessary to specify how its aggregated remote service operations should be used to attend that invocation.

The description of the execution model of atomic tasks is presented in two levels. The first level, presented in Section 7.5.2 , describes how multiple remote services are coordinated during the execution of a single task instance. The second level, presented in Section 7.5.4, describes the necessary steps to coordinate each remote service invocation.

7.5.1 The Mediator Service Execution

An execution of a mediator service operation is, actually, a transactional execution of instances of its aggregated remote services. Such execution is coordinated by atomic tasks instances and is done in two steps. First, the atomic task instance generates a plan for scheduling remote service operations. This plan is named *mediator plan* and uses a subset of the remote service operations aggregated by the mediator service operation that implements the atomic task. The next step is the execution of such mediator plan.

A mediator plan is a special kind of composite task specification in which its component tasks are *remote services*. Similarly, to composite tasks specifications, mediator plans model remote service as well as its associated wiring by means of control links and data links. An execution of a mediator plan consists of creating all its component remote service instances, passing input parameters to these instances,

determining the start remote service instances and performing them, receiving output of completed instances, determining their actual successors, materializing their input, performing them, and so on.

The execution of an atomic task instance ends when the scheduling of its plan finishes. At this point, if at least one remote service instance commits then the atomic task instance is committed. On the other hand, if all remote service instances fail then the atomic task instance is aborted.

7.5.2 Atomic Task Instances

The execution of mediator service operations is coordinated by instances of atomic tasks. A task instance is responsible for receiving invocations from the task scheduler, generating a mediator plan for such invocations, executing that plan, and returning the results back to the task scheduler.

The execution of a mediator service operation starts when the task scheduler sends a `submit` message to an atomic task instance. Upon receiving that message, the task instance starts the first phase of the execution of the mediator service operation, the *selection phase*.

The goal of the selection phase is to determine the set of remote services that will be used to accomplish the task. This set is a subset of all remote services aggregated by the mediator service that implements that task instance. Such selection is based on two criteria. The first is the **content description** of remote services. The second is the supported **transaction behavior** of remote services.

The content description is used to determine, among the remote services aggregated by a mediator service, those capable of handle a given invocation. The task instance selects only those remote services whose content description matches the value of its real input parameters. Therefore, only remote services whose content description matches the associated real input parameters are selected to the next step.

After selecting the remote service, based on the content description criteria, the task instance applies the supported transaction behavior criteria. This criterion is based on the mandatory-compensating constraint that is set by the task scheduler, when it invokes the task instance. If the value of such constraint is set to true then only compensable remote

service operations can be selected. Otherwise, all available remote services operations are selected. When the task scheduler sets the value of the mandatory-compensating constraint to true, it tells the task instance to be prepared to receive a `compensate` message in a future time. Since only compensable operations support compensation, only they can be used to attend such invocations. On the other hand, when the task scheduler sets the value of the mandatory-compensating constraint to false, it tells the task instance that once it commits it will never be compensated for. Therefore, non-compensable operations are safe to attend such invocations. The remote services that pass through the selection criteria described above are the input data for the next phase.

After the selection phase, the task instance starts the *ordering phase*. This phase consists of generating the *mediator plan*, which is the specification of the execution order in which the remote service instances, chosen in the selection phase, must run. Such specification is similar to the composite task specification, using the same components to represent it, that is, control links, data links, and tasks (remote services).

The algorithm that generates the mediator plan receives as input a set of remote service and returns as output a directed graph, whose nodes are remote services and whose edges are control links and data links. Such control links represent a subset of the execution dependencies defined in Section 6.1.2. Only *weak_commit-start* and *abort-start* execution dependencies are used. The *weak_commit-start* dependency is used to build groups of remote service instances that run concurrently, while the *abort-start* execution dependency is used to build contingency paths employed whenever a previous executed group fails. Since remote service operations may have dissimilar transaction capabilities, it is necessary to guarantee that the ordering algorithm generates only *safe* schedules.

The notion of safe scheduling of remote service operations is the same as that used for composite tasks (Section 7.2). A scheduling of remote service operations is safe if, at any time, it is possible to guarantee that its execution will terminate without leaving any side effects. For example, a scheduling where two pivot remote service operations are allowed to execute concurrently is not safe. If both operations commit, it is not possible to undo the effects of neither one of them. Therefore, such scheduling must be avoided. On the other hand, if both operations are compensable, they can be executed concurrently. If both operations commit then the scheduler must pickup one and compensate for the other.

In order to generate only safe schedules, the ordering algorithm only allows concurrent execution of compensable remote services. Non-compensable remote services are always scheduled to run sequentially. According to these constraints, the ordering algorithm generates dependency graphs as follows:

- The set of compensable remote services is split in many subsets according to some configuration parameter. For instance, an optimization parameter can determine the number of remote services to be grouped together. Such parameter can establish a range varying from all compensable remote services to only one compensable remote service per group. When all remote services are grouped together, the execution is optimized for response time. On the other hand, when only one remote service is inserted in a group, the execution is optimized for resource consumption. All remote services of a group are scheduled to run concurrently. This behavior is specified through the alternative constructor (Section 6.4.1). Each remote service of a group becomes the synchronized tasks of the alternative constructor and one synchronization task is inserted for synchronizing the concurrent execution of those remote services. Each control link leaving the remote services and reaching the synchronization task is labeled with a *weak_commit-start* execution dependency. All remote services are marked as non-mandatory, while the synchronization task is marked as mandatory. Such alternative constructor guarantees the safe execution of each remote services group. Inside a group, only one remote service is allowed to commit, this is guaranteed by the navigation algorithm (**Algorithm 2**).

- Since non-compensable remote services must be scheduled sequentially, they are split in groups of only one remote service. Therefore, each non-compensable remote service becomes a whole execution path that has one task (the non-compensable remote service), which is marked as mandatory.

- Groups of remote services (both non-compensable and compensable) are wired together through control links representing the *abort-start* dependency. Such specification defines that a group of compensable remote services defines the contingency plan regarding its preceding execution group. This procedure is repeatedly applied until all groups are inserted in the execution dependency graph. We are not using any criterion to establish the preference order between

two groups of remote services. However, this is a typical case where a cost function must be applied. Depending on the active configuration parameter, the remote services can be ordered by resource consumption or response time, before starting its grouping.

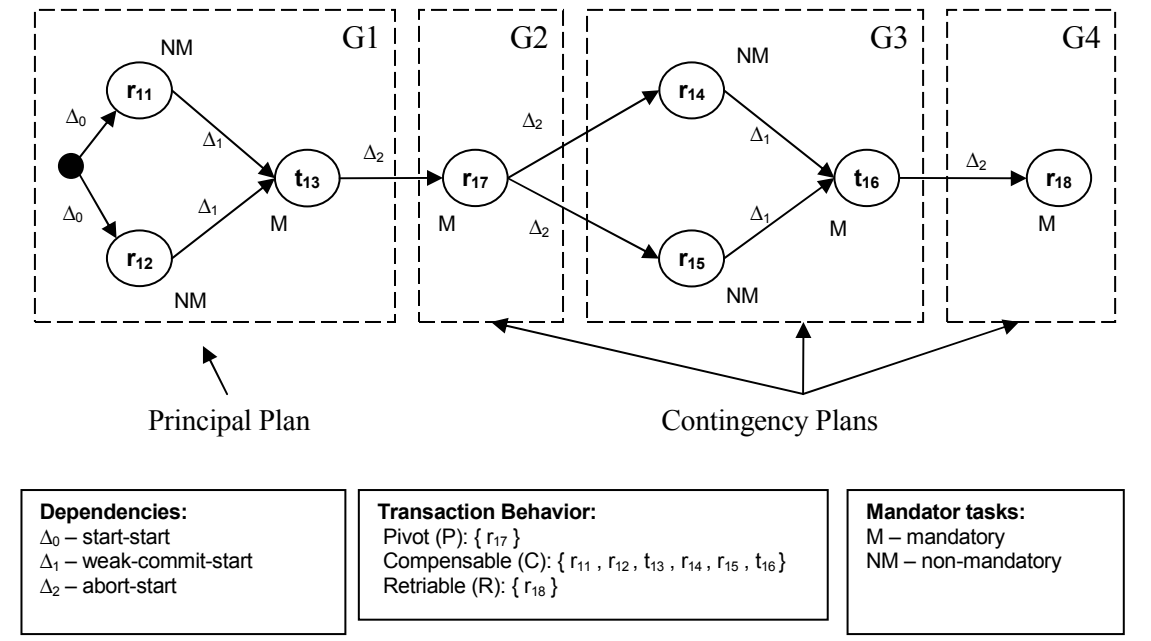


Figure 7.7 - Example of a mediator plan.

For example, consider a mediator service operation the aggregates six different remote service operations. Suppose that among those operations, four are compensable, one is pivot, and the last is retriable. Consider that such mediator service operation is invoked during a composite task execution. Assume that the mandatory-compensating constraint is set to false and that the content description of all remote service operation matches the real input parameter of the invocation. In this case, the selection phase must return all six remote services, which in turn will be used as input to the ordering phase. Suppose that the configuration parameter for grouping remote services is such that compensable operations must be grouped into sets containing fifty per cent of its total number. In this case, since there are four compensable operations, they will generate two groups of two remote services each. Considering no criterion to define the preference order among groups of services, one possible mediator plan is shown in Figure 7.7. There are four groups of remote services instances. Groups `G1` and `G3` are formed by

compensable remote service instances r_{11}, r_{12}, r_{14}, r_{15}, r_{16}, while groups G2 and G4 are formed by non-compensable remote service instances r_{17}, r_{18}.

Each group of compensable remote service instances are wired through an alternative constructor. For instance, group G1 has two synchronized tasks, remote service instances r_{11} and r_{12}, which are wired through weak_commit-start control links with the synchronization task t_{13}. This group configures the main execution path $\prec_{-ep}$, since the remote service instances r_{11} and r_{12} are wired through a start-start control link with the empty task. In that execution path, only the synchronization task t_{13} is marked as mandatory thus $\prec_{1\ -ep}$ succeeds if t_{13} succeed. A synchronization task, which is the target task of weak_commit-start control links, succeed if at least one of its synchronized tasks succeed, otherwise it fails. Therefore, $\prec_{1\ -ep}$ will succeed if either r_{11} or r_{12} commit.

The groups of non-compensable remote service instances are formed by only one non-compensable remote service instance, which is always marked as mandatory. For example, group G2 is formed by the pivot remote service r_{17}. This group also configures an execution path, in this case, a contingency path $\prec_{1C_1\ -ep}$ regarding $\prec_{1\ -ep}$. The contingency path is executed only if $\prec_{1\ -ep}$ fails and after its execution it succeeds if the remote service instance r_{17} commit. Otherwise, $\prec_{1C_1\ -ep}$ fails. The structure and behavior of the other groups follow the same structure and behavior described for groups G1 and G2.

Clearly, the mediator plan of Figure 7.7 is safe. Since all non-compensable remote service instances are executed sequentially and all concurrently executed remote service instance are compensable and structured using the alternative constructor, all execution paths are guaranteed to be executed without leaving any side effects. Furthermore, the algorithm for ordering remote service instances always generates dependency graphs that ensure the well-formed structure of **Definition 13**.

The next phase of the execution of mediator service operations is the *scheduling phase*. This phase is performed by the *mediator service scheduler*. The scheduling phase employs the same algorithm used for scheduling task instances. Actually, the mediator

service scheduler is simply an instance of the task scheduler that enforces inter-remote-service dependencies according to the mediator plan, generated in the ordering phase.

7.5.3 Algorithms for Coordinating the Execution of Atomic Task Instances

Atomic tasks implement the operations `create`, `submit`, `abort`, `getTransactionBehavior`, and `terminate`, which are defined by the transactional lifecycle interface. These operations implement all necessary coordination for the execution of atomic task instances. In this section, we only present the algorithm of the `submit` operation. Due to their simplicity, the algorithms of the operations `create`, `abort`, `getTransactionBehavior`, and `terminate` are not shown. However, the following paragraph contains a brief explanation of how such operations are implemented.

The `create` operation instantiates a new task instance and returns its `id` as a result. The `abort` operation tells a task instance to abandon the execution of a running `submit` operation. When a task instance receives a call on such operation, it sends an `abort` message to all remote service instances of its mediator plan whose execution state is *committed*. The `getTransactionBehavior` operation returns the transaction behavior of the task instance, which is based on the transaction behavior of all remote service instances of its mediator plan. When a call on such operation is received, the task instance sends a `getTransactionBehavior` message to all of the remote service instances of its mediator plan. If at least one remote service instance returns *compensable* then the task instance also returns *compensable*. Otherwise, if at least one remote service instance returns *retriable* then the task instance also returns *retriable*. Finally, if all remote service instances return *pivot* then the task instance also returns *pivot*. The `terminate` operation tells a task instance to finish any pending work and to release the system resources used by it. When a task instance receives a call on such operation, it sends a `terminate` message to all remote service instances of its mediator plan.

Algorithm 6 defines the submit operation. This algorithm makes use of two other algorithms, the selecting phase algorithm and the ordering phase algorithm, which are presented next.

Algorithm 6 (Submit Operation)
Variables of the Task Instance:

`Svr`, *the WSTL id of the mediator service that implements* `Op`;

`Op`, *the operation that implements the Task instance;*

`InputParam`, *input parameters of* `Op`.

`OutputParam`, *output parameters of* `Op`.

`FaultParam`, *fault parameters of* `Op`.

`committedRS`, *the id of the remote service instance that remains in the committed state after the scheduling phase, if any.*

Input Parameters

`Mc`, *the mandatory-compensate constraint.*

1. *Select the mediator service operation* `MSop` *(from the mediator repository) such that mediator service id =*`Svr` *and operation =* `Op`.
2. *For each value* `v` *of the parameters of* `InputParam` *do:*

2.1 If `v` *does not belong to the content description of* `MSop`, *then set the state of this task instance to aborted and resume to (8).*

3. *Start the Selecting Phase (Algorithm 7) passing* `Mc`.
4. *Start the Ordering Phase (Algorithm 8) passing the remote services selected in (3).*
5. *Start the Scheduling phase (Algorithm 2) passing the dependency graph* `D(ST)` *generated in (4).*
6. *If (5) return* `C(ST)` *then:*

6.1 Store in `committedRS` *the id of the committed remote service instance.*

6.2 Set the state of this task instance to committed.

7. *Otherwise:*

7.1.If this task instance is retriable then resume (5)

7.2 Otherwise, set the state of this task instance to aborted.

8. *Return.*

Next, **Algorithm 7** shows the steps for selecting the remote service providers used to compose a mediator plan.

Algorithm 7 (Selecting Phase)
Variables of the Task Instance:

`Svr`, *the WSTL id of the mediator service that implements* `Op`;

`Op`, *the operation that implements the Task instance;*

`InputParam`, *input parameters of* `Op` *in the mediator service format.*

`OutputParam`, *output parameters of* `Op` *in the mediator service format.*

`FaultParam`, *faultOutput parameters of* `Op` *in the mediator service format.*

Input Parameters

`Mc`, *the mandatory-compensate constraint.*

Output Parameters

`Q(RSop)`, *a queue containing the set of selected remote service operations.*

1. Select all remote services operations `RSop` *(from the mediator repository) such that its aggregating mediator service id =* `Svr` *and its aggregating mediator service operation id =* `Op`.

2. For each remote service operation `RSop` *do:*

2.1 Determine the supported transaction behavior `Tb` *of* `RSop`.

2.2 If the values of all parameters in `InputParam` *belongs to the content description of* `Rsop` *then*

(a) If `Mc` *is true then*

(i) If `Tb` *is compensable or virtual-compensable then insert* `RSop` *in* `Q(RSop)`.

(b) Otherwise, insert `RSop` *in* `Q(RSop)`.

3. Return `Q(Rsop)`.

Next, the algorithm for generating a mediator plan.

Algorithm 8 (Ordering Phase)

Configuration Variables:

`GrpFactor`, *the grouping factor, used to generate the groups of remote service instances of a mediator plan;*

Input Parameters

`Q(RSop)`, *a queue containing the set of selected remote service operations.*

Output Parameters

`D(ST)`, *a dependency graph representing the execution order of a set of remote service instances.*

1. `CompRSop` = *Select all remote services operation* `RSop` *in* `Q(RSop)`, *such that* `Rsop` *is compensable.*

2. `NCompRSop` = *Select all remote services operation* `RSop` *in* `Q(RSop)`*, such that* `Rsop` *is non-compensable.*
3. *Order* `CompRSop` *by some cost function.*
4. *Split* `CompRSop` *in execution groups according to* `GrpFactor`.
5. *Put each element of* `NCompRSop` *in a separated execution group.*
6. *Order the execution groups by some cost function.*
7. *Generate a dependency graph* `D(ST)` *according to the following rules:*

7.1 Generate a remote service instance for each element in the execution groups.

7.2 Assign each `RSop` *in the execution groups to the remote service instances generated in (7.1).*

7.3 For each execution group `Grp` *do:*

(a) If `Grp` *has more than one remote service instance then create a new synchronization task and connect it to all remote service instances of* `Grp` *through weak_commit-start control links, such that the synchronization task becomes the target task of these control links .*

7.4 Select all remote service instances assigned to the first execution group.

7.5 Create an empty task and connect it to the remote service instances of (7.3) through start-start control links, such that the empty task becomes the source task of these control links .

7.6 For each execution group `Grp` *do (following their ordering):*

(a) If `Grp` *has more than one remote service instance then connect its synchronization task to all remote service instances of the next execution group through abort-start control links, such that the synchronization task becomes the source task of these control links.*

(b) Otherwise, connect the remote service instance of `Grp` *to all remote service instances of the next execution group through abort-start control links, such that the remote service instance of* `Grp` *becomes the source task of these control links.*

3. *Return* `D(ST)`.

Algorithm 8 can generate dependency graphs containing retriable remote services. By definition, retriable remote services always commit after a finite number of tries. Based on such definition, one can conclude that it is meaningless to have any kind of remote service following a retriable remote services in a given mediator plan. Since

retriable remote services always commit, a contingency plan following it will never be reached. However, this conclusion is not true in the execution model of WebTransact. Whenever a retriable remote service is participating in a mediator plan, it is treated as a regular service, i.e., it is invoked only once, despite its success or fail. The retriability property is addressed at the level of the task instance rather than the remote service instance level (**Algorithm 6**).

7.5.4 Remote Service Instances

Remote services are responsible for receiving invocations from the mediator service scheduler and for coordinating the execution of remote service operations. Therefore, the remote service has to support the task lifecycle interface defined in Section 7.1.3. Therefore, it provides operations for instantiating, terminating, and for controlling the state transitions of a given remote service instance. When the mediator service scheduler receives an instance of a mediator plan to be executed, it calls the operations defined in the lifecycle interface, and implemented by remote services, to control the scheduling of the remote service instances.

When an instance of a remote service receives a `submit` message, it first has to build the input element of the target operation from the input parameters received along the `submit` message. Such input parameters are in the mediator service format thus they must be converted to the specific format of the target remote service. This is done applying mapping information to that input parameters. Such mapping information is retrieved from the WSTL definitions of the remote service. The input message produced above is not in the final format to be send to the remote service provider. Recall from Section 5.2 that each WSDL port type is mapped to a WSTL remote service. Therefore, the input message of a given remote service operation represents the input element of a given port type operation. Since WSDL port types are abstract definitions, the produced message is an abstract message. In order to produce the concrete message, first it is necessary to select an endpoint to send it. This endpoint is selected among the available WSDL port elements associated with the remote service. Recall from Section 2.6.4 that one WSDL port type might have many WSDL bindings, which, in turn, might be referenced from many WSDL ports. Therefore, one remote service operation can be associated to many different WSDL ports, i.e., the same operation can be invoked through different physical addresses. Once an endpoint is selected, the remote service generates

the concrete message using the abstract message and the binding information available. This concrete message is then sent to the remote service provider through the selected endpoint. Note that we are considering that the WebTransact system supports all types of bindings and there is no preference between them. At this point, the remote service instance waits for the response of the remote service provider. Upon receiving the result, the remote service instance unpacks the returned message, converts its content to the mediator service format, and fills out the output or fault parameters accordingly to the WSTL mapping information. If the returned message generates output parameters then the state of the remote service instance is set to *committed*. On the other hand, if a fault message is generated then the state is set to *aborted*. After having set its state, the remote service instance returns back to the mediator service scheduler. If the call to the remote service provider fails - for instance, due to a time-out - the remote service instance can repeatedly try another available endpoint. If all endpoints fail, the remote service instance builds a fault message, sets its state to *aborted*, and returns the control back to the mediator service scheduler. From this point on, two different events can occur:

- The remote service instance can receive a `compensate` message from the mediator service scheduler. Once received that message, the remote service instance starts its *recover procedure*. First, the remote service queries the mediator repository to determine the compensating operation to be used. After that, it prepares a message to be sent to the remote service provider, in order to compensate the previous executed operation. Such message is prepared and invoked through the same steps used to invoke regular operations, as described in the previous paragraph. Since compensating operations are retriable, the remote service instance repeatedly invokes the remote service provider until an invocation succeeds. The recover procedure is performed only by remote service instance supporting the compensable transaction behavior. If a non-compensable remote service receives a `compensate` message, it returns an error.

- The remote service instance can receive a `terminate` message from the mediator service scheduler. Once received that message, the remote service instance ceases to exist. Both compensable and non-compensable remote services must receive a `terminate` message. Such message is necessary to tell

the remote service instance that it is safe to destroy itself for releasing system resources.

The above procedures for calling remote service providers consider remote service operations that do not have local transaction support. This includes pivot, retriable and compensable remote service operations. For virtual-compensable operations, the procedures are slightly different. When a remote service instance receives a `submit` message on a virtual-compensable operation, it first verifies the mandatory-compensate constraint that came along to the `submit` message. If the value of such constraint is true then the remote service provider must be invoked under a transactional context. This is necessary because virtual-compensable operations are compensated through an abort of a transaction instead of the execution of another operation. To invoke the remote service provider under a transaction context, the remote service instance communicates with the transaction coordinator of the WebTransact system (Section 4.2.9.1) and asks it to start the coordination of a new transaction on behalf of that invocation. The mediator transaction coordinator will then be responsible to communicate with the transaction coordinator of the remote service provider and to run the TIP protocol (Section 4.2.9.3) for that invocation. The WSTL mapping information on transaction coordinators (Section 4.2.9.4) is used to discover the physical location and message format to communicate with transaction coordinator of the remote service provider. The mediator transaction coordinator can use either the pushing or the pulling models (Section 4.2.9.3) to start a new transaction context. After creating the transaction context, the mediator transaction coordinator sends a TIP transaction identifier to the remote service instance. This transaction identifier is then packed along the other contents and sent to the remote service provider. Upon receiving the result, the remote service instance unpacks the returned message, converts its content to the mediator service format, and fills out the output or fault parameters accordingly to the WSTL mapping information. At this point, the following events can occur:

- If the returned message generates output parameters then the remote service instance asks the mediator transaction coordinator, using the transaction identifier of the current transaction, to start the first phase of the two-phase commit for that transaction. Then, the mediator transaction coordinator sends a `prepare` message, along the TIP transaction identifier, to the transaction

coordinator of the remote service provider. When the mediator transaction coordinator receives the response, two different events can occur:

- If the mediator transaction coordinator receives a `prepared` message, the remote service instance sets its state *committed* and returns the control back to the mediator service scheduler. The transaction remains in the *prepared* state until the remote service instance receives either a `terminate` or a `compensate` message from the mediator service scheduler. If a `terminate` message is received then the remote service instance asks the mediator transaction coordinator to commit the transaction. Therefore, the mediator transaction coordinator starts the second phase of the two-phase commit to the transaction. At this point, the remote service instance ceases to exist.

- If the mediator transaction coordinator receives an `aborted` message, the remote service instance builds a fault message and returns the control back to the mediator service scheduler. In this case, the response received from the remote service provider is discarded. When the remote service instance receives a `terminate` message from the mediator service scheduler, it ceases to exist.

- On the other hand, if the message returned by the remote service provider generates a fault message then the state of the remote service instance is set to *aborted* and the remote service instance asks the mediator transaction coordinator to abort the transaction. At this point, the transaction context created to coordinate the invocation is destroyed. From now on, the remote service instance waits for a `terminate` message from the mediator service scheduler. When such message arrives, the remote service instance ceases to exist.

The protocol from synchronizing transaction follows the two-pipe model (Section 4.2.9.3). The transaction synchronization messages are separated from the application messages. The endpoint used by the mediator transaction coordinator to communicate with the transaction coordinator of the remote service provider is completely independent from the endpoint used by the remote service instance to invoke messages on the remote

service provider. The messages sent to those endpoints can even be produced using different WSDL bindings thus, they can be transported using different protocols.

7.5.5 Algorithms for Coordinating Executions of Remote Service Instances

In this section, we present the algorithms used by remote service instances to coordinate the transactional execution of remote service operations. These algorithms are the implementations of the operations of the lifecycle interface. First we show the algorithm of the `submit` operation. Next, an auxiliary algorithm is presented, the invoke algorithm. Then the algorithm that implements the `compensate` operation is described.

Due to its simplicity, the algorithms of the operations `create`, `abort`, `getTransactionBehavior`, and `terminate` are not shown. The `create` operation instantiates a new remote service instance and returns its `id` as a result. The `abort` operation tells a remote service instance to abandon the execution of a running `submit` operation. When a remote service instance receives a call on such operation, it waits for the response of any pending invocation to a remote service provider. When such invocation returns, if it fails, the remote service instance only has to set its state to aborted and nothing more. However, if that invocation succeeds then the remote service instance has to compensate for it. This is done through a self-call to the `compensate` operation. The `getTransactionBehavior` operation returns the transaction behavior of its assigned remote service operation (retrieved from the WSTL definitions stored in the mediator repository). The `terminate` operation tells a remote service instance to finish any pending work and to release the system resources used by it. Actually, only remote service instances assigned to virtual-compensable operations have pending work to do. When such operation receives a `terminate` message, if its execution state is set to committed, it asks the mediator transaction coordinator to start the second phase of the 2PC protocol on behalf of the transaction assigned to that remote service instance.

Algorithm 9 (Submit Operation)
Variables of the Remote Service Instance:

`Svr`*, the WSTL id of the remote service that implements* `Op`*;*

`Op`*, the operation that implements the Remote Service Instance;*

`InputParam`*, input parameters of* `Op` *in the mediator service format.*

`OutputParam`*, output parameters of* `Op` *in the mediator service format.*

`FaultParam`*, input parameters of* `Op` *in the mediator service format.*

`State`*, the current execution state of the remote service instance.*

1. *If* `Op` *is a virtual-compensable operation then ask the Mediator Transaction Coordinator for a new TIP transaction identifier* `TXID` *(passing* `Svr` *and* `Op` *along the call).*
2. *Call the Invoke operation passing* `Svr`*,* `Op`*,* `InputParam`*,* `OutputParam`*,* `FaultParam` *and,* `TXID` *(Algorithm 10).*

3. If the invoke operation returns true then

3.1 If `OutputParam` *is not empty then:*

(a) If `Op` *is a virtual-compensable operation then:*

(i) Ask the Mediator Transaction Coordinator to start the first phase of the 2PC protocol on behalf of transaction `TXID`.

(ii) If the Mediator Transaction Coordinator returns prepared then resume (7).

(iii) Otherwise, generate a fault message "SERVICE ABORTED BY THE REMOTE SERVER" write `FaultParam` *with this fault message. Clear the contents of* `OutputParam`.

(iv) `State` = *aborted. Resume to (8).*

(b) `State` = *committed.*

4.2 If `FaultParam` *is not empty then:*

(a) If `Op` *is a virtual-compensable operation then:*

(i) Ask the Mediator Transaction Coordinator to abort transaction `TXID`.

(b) `State` = *aborted.*

5. *Otherwise, generate a fault message "NO RESPONSE FROM REMOTE SERVER" and write* `FaultParam` *with this fault message.*
6. *Return.*

Algorithm 10 presents the steps to invoke an operation of a remote service provider. Such algorithm makes use of all available ports specified for that operation. If the remote service provider replies in one of those tries, then a Boolean value true is returned. Otherwise, a Boolean value false is returned.

Algorithm 10 (Invoke Operation)
Input/Output Variables:

`Svr`*, the WSTL id of the remote service that implements* `Op`*;*

`Op`*, the operation that implements the Remote Service Instance;*

`InputParam`*, input parameters of* `Op` *in the mediator service format.*

`OutputParam`*, output parameters of* `Op` *in the mediator service format.*

`FaultParam`*, input parameters of* `Op` *in the mediator service format.*

`TXID`*, a transaction id of the current transaction (only when* `Op` *is a virtual-compensable operation).*

Output:

`Reply`*, a Boolean value indicating that a remote service provide reply a given invocation;*

1. `RS` = *Select the WSTL definitions (from the mediator repository) of the remote service operation such that remote service id =*`Svr` *and operation =* `Op` .
2. *Generate the abstract input message* `AbstInputMsg` *applying the input parameter mapping of* `RS` *to* `InputParam`.
3. `Ports` = *Select all available WSDL ports associated to the WSDL portType element referenced by* `RS`.
4. `Reply` = *false.*
5. *While exist elements in* `Ports` *do:*

5.1 `Port` = *Select one WSDL port element of* `Ports`.

5.2 `Binding` = *Select the binding definition of* `Port`.

5.3 Generate the concrete input message `ConcInputMsg` *applying* `Binding` *to* `AbstInputMsg`. *If* `Op` *is a virtual-compensable operation then include* `TXID` *in* `ConcInputMsg` .

5.4 Send `ConcInputMsg` *to the physical address defined in* `Port` *using the transport protocol defined in* `Binding`.

5.5 Upon receiving the response:

(a) Unpack it using `Binding`.

(b) Apply the WSTL parameter mapping of `Op` *to the unpacked message.*

(c) `Reply` = *true.*

5.6 If the time-out is reached and no reply from the remote service provider is received then resume to (5).

6. Return `Reply`.

Next, the algorithm used to compensate for the work done by remote service providers.

Algorithm 11 (Compensate Operation)
Variables of the Remote Service Instance:

`Svr`*, the WSTL id of the remote service that implements* `Op`*;*

`Op`*, the operation that implements the Remote Service Instance;*

`InputMsg`*, abstract input message of* `Op` *in the remote service format.*

`OutputMsg`*, abstract output message of* `Op` *in the remote service format.*

`State`*, the current execution state of the remote service instance.*

`TXID`*, the transaction id of the current transaction.*

`CompOper`*, the compensating operation of* `Op`*.*

`CompSvr`*, the WSTL id of the remote service that implements* `CompOper`*;*

1. *If* `State` ≠ *committed then raise error "INSTANCE NOT IN COMMITTED STATE". Return.*
2. *Determine the transaction behavior* `Tb` *of* `Op` *using the WSTL definitions (from the mediator repository) of the remote service operation such that remote service id =* `Svr` *and operation =* `Op`*.*
3. *If* `Tb` *does not support compensation then raise error "OPERATION UNSUPPORTED". Return.*
4. `State` = *compensated. Return the control to the caller and resume next.*

3. If `Tb` = *virtual-compensable then:*

3.1 Ask the Mediator Transaction Coordinator to abort the transaction `TXID`*.*

4. Otherwise:

4.1 If `Tb` = *is compensable by an active action (Section 4.2.5) then:*

(a) Determine the compensating operation `CompOper` *of* `Op` *using the WSTL definition* `Tb`*.*

(b) Call the Invoke operation (Algorithm 10) passing `CompSvr`*,* `CompOper`*,* `InputMsg`*,* `OutputMsg`*,* `FaultParam (= Null)` *and,* `TXID (= Null)`*.*

(c) If the invoke operation return false then resume (a).

7.6 EXTENDING THE GUARANTEED-TERMINATION PROPERTY

We have seen in Section 7.3.5 that composite tasks having well-formed structure ensure the *guaranteed-termination* [5] property. The guaranteed-termination property of transactional composite tasks is a generalization of the *all-or-nothing* semantics of traditional ACID transactions. Since a composite task CT_i can have multiple execution paths $\prec -eps$, CT_i can commit even when some of its tasks have aborted as long as all mandatory tasks in one $\prec -eps$, say $\prec -ep_c$, terminate successfully. In this case, the completion $C(CT_i)$ requires that all tasks of CT_i, which do not belong to $\prec -ep_c$, be in state: *not-executed*, *aborted*, or *compensated*. The guaranteed-termination property is a correct criterion that ensures the failure atomicity at the composite task level. However, as described in Section 7.5, the transaction model of WebTransact has another level of execution, the execution level of atomic tasks. We have seen in Section 7.5 that an execution of an atomic task is, in fact, and execution of a composite task whose component tasks are implemented by remote service operations instead of mediator service operations. Therefore, an invocation of an atomic task triggers the execution of another complex transaction implemented by the scheduling of remote service operations (according to the mediator plan). This leads to a transaction model with one level of nesting, where an atomic transaction execution is a subtransaction of the transactional execution of its composite task. Similarly to composite tasks, an atomic task AT_i can have multiple execution paths $\prec -eps$, hence AT_i can commit even when some of its remote service instances have aborted as long as all one of its remote service instances in one $\prec -eps$, terminate successfully. Moreover, the scheduling of remote service instances ensures that, if $C(AT_i)$ then only one remote service instance of AT_i is in the committed state while all other remote service instances are in the executing states: not-executing, aborted, or compensated. Therefore, the scheduling of remote service operations ensures the guaranteed-termination property at the atomic task level. Hence, the transaction model of WebTransact not only guarantees the failure atomicity at the composite task level, but it also guarantees the failure atomicity at a lower level, the atomic task level. The *2L-guaranteed-termination* (two-level-guaranteed-termination) property encompasses this notion of layered failure atomicity ensured by the transaction model of WebTransact.

Theorem 2

A well-formed execution CT_i of a correct composite task specification CTS_i ensures the 2L-guaranteed-termination property.

Argument

From Theorem 1, we know that well-formed executions of a correct composite task specification ensure the guaranteed-termination property. An atomic task execution is a scheduling of remote service operations according to the mediator plan. Since the algorithm for ordering remote service instances (Algorithm 8) in a mediator plan always generates dependency graphs that ensure the well-formed structure of Definition 13, the scheduling of remote services is equivalent to a well-formed composite task execution. Therefore, the scheduling of remote service instances ensures the guaranteed-termination property at the execution level of atomic tasks. Thus, a well-formed execution CT_i of a correct composite task specification CTS_i ensures the guaranteed-termination property at both the composite task level and the atomic task level, hence CT_i ensures the 2L-guaranteed-termination property.□

8. Conclusions

The Web services technology enables a new business opportunity where one company can offer value-added services to its customers through the composition of basic Web services. However, the current Web services technology solves only part of the problem of building Web services compositions. Building reliable Web services compositions requires much more than just addressing interoperability between client programs and Web services. Besides interoperability, building Web services compositions requires mechanisms for: describing the dissimilar transaction support of Web services, resolving the semantic and content heterogeneity of semantically equivalent Web services, specifying the transaction interaction patterns among Web services, and coordinating such interaction patterns. Due to this novel set of requirements, posed by the Web services environment, the existing business process frameworks cannot be directly applied to develop Web services compositions. Therefore, there is a need of new frameworks, specifically developed for addressing the new requirements of the Web service environment. The main contribution of this work is the specification of one of such frameworks, the WebTransact framework.

WebTransact addresses the requirements for building reliable Web services compositions. WebTransact treats the problem of building composition in an integrated way, providing mechanisms for describing the dissimilar transaction behavior of Web services, for aggregating semantically equivalent Web services and for resolving their heterogeneities, for specifying reliable interaction patterns of Web services, and for coordinating such interaction patterns in a transactional way. To the best of our knowledge, there is no other works on integrated frameworks addressing all the requirements addressed by WebTransact.

8.1 CONTRIBUTIONS

The first contribution of this work is the description of an integrated view of the problem of building Web services compositions. Few works ([8], [76], [129], [136]) deal with the problem of building Web services compositions. All these works concentrate their efforts on a subset of the requirements presented in this work. For instance, the work

presented in [76] is concentrated on the problem of defining interaction patterns of Web services but do not consider the problem of supporting Web services with dissimilar transaction behavior. On the other hand, the work presented in [129] contemplates that problem but neither considers a transaction model for supporting its proposal nor considers the problem of aggregating semantically equivalent Web services. In Chapter 2, we have presented a more generalized view of the problem of building value-added services through the composition of basic Web Services. Such generalized view can be used as a basic set of issues for the development of other frameworks for composing Web services.

The second contribution is the definition of a multilayered architecture for addressing the problem of heterogeneity and dynamism of the Web service environment. Such architecture is based on the technology of Wrapper mediator systems ([19], [42], [50], [51], [75], [104], [128], [132], [134], [135], [150]). However, the WebTransact architecture addresses *Web service* mediation while Wrapper mediator systems address *information* mediation. The WebTransact architecture has a set of specialized components to handle Web service mediation. The *Remote service* component resolves conflicts involving the dissimilar representation of knowledge of different Web services, and conflicts due to the mismatch in the content capability of each Web service. Besides resolving structural and content conflicts, *remote services* also provides information on the interface and the transaction behavior supported by Web services. The *Mediator service* component aggregates semantically equivalent remote services providing a homogenized view on heterogeneous Web services. Finally, Web services compositions are built on top of those mediator services. The WebTransact architecture provides the necessary isolation between the Web service providers and the Web services composition. Since compositions are built upon mediator service operations, changes in the behavior, content, or interface of Web services as well as the aggregation of new Web services do not directly influence compositions. Such layer of mediation provides a very efficient mechanism to deal with the heterogeneity and dynamism of the Web service environment.

The third contribution is the definition of the Web Service Transaction Language (WSTL). WSTL provides elements for defining the supported transaction behavior of Web services, for resolving semantic and content dissimilarities of Web services, for aggregating semantically equivalent Web services, for defining mediator service interfaces, and for defining reliable interaction patterns of Web services composition. The

WSTL *transaction behavior* element supports the definition of different levels of transaction support, ranging from no transaction support to systems that have full support for distributed transactions. Therefore, a wide variety of Web services supporting different kinds of transaction interactions can be integrated in WebTransact. The WSTL *remote service* element supports the aggregation of semantically equivalent Web services providing mechanisms for resolving the semantic and content dissimilarities of Web services. The WSTL *mediator service* element defines a homogenized and integrated view of semantically equivalent Web services that is used to build compositions. The WSTL elements: *composite task*, *atomic task*, *execution dependency*, *data link*, *rules*, and *mandatory tasks* support the specification of transaction interaction patterns. WSTL supports the recursive definition of composite tasks, thus more complex interaction patterns can be defined through the composition of existing ones. The execution dependency element provides support for defining different levels of reliability and optimization of Web services composition. Through the *abort-start* execution dependencies, a composition developer can define contingency paths to improve the reliability of a given composition. Through the *alternative constructor,* a composition developer can define different patterns of concurrently executing services to improve the response time of a given composition. Finally, WSTL is an extension of the Web Service Description Language (WSDL) [137] that is adherent to XML-based standards that enable the Web service technology. Due to this feature, WSTL can be used as a complementary standard for specifying transactional compositions of Web services. Therefore, client programs of WSTL compositions can be either an application program or another system that is able to interact with a regular Web service.

The fourth contribution of this work is the definition of a flexible framework for integrating Web services that have (and expose) their own local transaction support based on the two-phase commit protocol (2PC). The key concepts of such framework is the use of a two-pipe model for communicating transaction synchronization messages and the use of the WSDL framework for exposing the 2PC communication interface of a given transaction manager. The WebTransact framework makes use of the Transaction Internet Protocol (TIP) ([35], [36]) for coordinating transaction synchronization messages between remote transaction managers and WebTransact. The TIP protocol supports the two-pipe model, thus satisfying the key requirement of transaction processing for Web services environments: the separation of the transaction synchronization messages from the

application communication protocol. To improve flexibility, the WebTransact framework does not enforce any standard interface for TIP commands. A remote transaction manager can expose its proprietary TIP interface as long as there is a WSTL *tmmap* element for that interface. Therefore, any transaction manager that supports the TIP protocol (or even any flavor of the presumed abort 2PC protocol) can define abstract operations through a WSDL interface, map these operations to the standard TIP commands, and then define how these abstract operations must be sent using WSDL binding definitions. Since TIP and WSDL are becoming well-accepted standards, this framework combines ubiquity with flexibility.

The fifth contribution is the definition and formalization of a transaction model and its related protocol for coordinating the execution of Web services compositions specified through WSTL. The transaction model of WebTransact exploits the dissimilar transaction behavior of Web services and guarantees the correct and safe execution of mediator compositions. The notion of correctness of composition execution is based on both the user needs (composition specification) and the *2L-guaranteed-termination* criterion. The *2L-guaranteed-termination* is a new correctness criterion suited for the Web services environment where the atomic property of the composition is relaxed to contemplate both the concept of compensating and non-mandatory operations and the potential existence of many semantically equivalent Web services.

8.2 FUTURE WORKS

The WebTransact framework can be used as the infrastructure of many other works. In the next paragraphs, we present some of the research areas that can be explored in the context of the WebTransact framework.

Implementation Architectures: There are interesting experimentations that can be done through the implementation of (parts of) the WebTransact framework. For instance, the mediator repository can be implemented using different types of database systems such as the relational [28], object-relational ([20], [121]), or the object-oriented DBMS ([24], [67]); another interesting option would be a native XML-based repository ([16], [25]). The components of WebTransact architecture can be implemented using distributed platforms like CORBA [97] and EJB [124]. This would improve the flexibility and scalability of WebTransact since such implementation would lead to a fully distributed

and decentralized architecture. Since WebTransact uses standard Web services technology only, the interoperability between WebTransact components and Web services can be implemented using existent Web services toolkits such as "IBM's Web services toolkit" [60] or ".Net framework SDK" [85]. The database group of PESC/COPPE[18] has already started the implementation of the WebTransact framework using the "IBM's Web services toolkit". This work is being done as part of a master thesis project. The first prototype (with a limited set of features) is expected to be implemented by the end of September 2002.

Distributed Transaction Coordination. We have adopted a two-pipe model along with the TIP protocol to coordinate distributed transactions involving Web services that expose the 2PC interface of its local distributed transaction coordinator. In order to facilitate the integration of different transaction processing systems it is important to investigate the issues of building TIP state machines for the most common distributed transaction processing systems such as OMG's Object Transaction Service (OTS) [96], Sun's Java Transaction Service (JTS) [127], and Microsoft's Distributed Transaction Coordinator (DTC) [86].

Dynamic Discovering of Services. In this work, we did not address the issue of discovering Web services [66]. The Web services are statically integrated in WebTransact by a developer who plays the role of Web services integrator. However, this is not a flexible way of doing Web services integration. Therefore, there is the need of developing (semi-) automatic tools to help the task of discovering and integrating Web services.

Supporting Tools. The task of building mediator compositions using WSTL elements is error prone and tedious. Therefore, there is a need of specific (graphic) tools to specify the control links and data links of a composition and to automatically build input, output, and fault messages of composite tasks. Such tools must also provide mechanisms for checking the correctness of composition specifications.

Security. In the same way as the transaction behavior of Web services, the requirements regarding security are likely to be dissimilar between Web services [115]. Therefore, we believe that the specific *security behavior* of Web services must be explicitly defined and mechanisms for mediating such dissimilar behavior must be

[18] Computer Science Engineering Program of Federal University of Rio de Janeiro (UFRJ).

developed. This is an interesting and important issue to be investigated in the context of the WebTransact framework.

Quality of Service. In the same way networked applications must be monitored to improve whole system quality ([53], [147]), it is necessary to audit and analyze Web services execution in order to improve the quality and efficiency of mediator service compositions.

Application Domains. The WebTransact framework has been developed to serve as basic infrastructure to build the business processes of the new era of Web services. However, there are other application domains where the ideas of WebTransact can be useful. For instance, it is important to investigate how the multilayered architecture of WebTransact can be exploited to aid the development of specific distributed computing applications such as in *Grid computing* ([18], [74]). According to [48], the Grid computing provides mechanisms to couple geographically distributed resources and it offers consistent and inexpensive access to resources irrespective of their physical location. It enables sharing, selection, and aggregation of a wide variety of geographically distributed computational resources (such as supercomputers, computer clusters, storage systems, and data sources). Since Grid computing is moving towards Web service technology ([57], [119], [122]), we believe that the WebTransact support for heterogeneous transactions can be seamlessly coupled to grid platforms.

Methodologies. WebTransact provides the necessary tools to build a new unit of software code, the Web services composition. Web services composition [102] has a higher granularity and embeds more semantics than a traditional server component such as a CORBA [97] component or a COM [84] component. There is a need to investigate the impact that Web services compositions will have on the existent component-based approaches for developing internet applications ([27], [81], [80]).

9. Acknowledgements

The authors would like to thank Prof. Louiqa Raschid for her valuable suggestions and discussions, during their joint work at the University of Maryland at College Park.

10. References

[1] Agrawal, D., Abbadi, A. E., “Transaction Management in Database Systems”. In: **[31]**, Chapter 1.

[2] Alonso, G., Fessler, A., Pardon, G., et al., “Correctness in General Configurations of Transactional Components”. In: *Proceedings of the Symposium on Principles of Database Systems (PODS)*, Philadelphia, Pennsylvania, pp. 285-293, 1999.

[3] Alonso, G., Fessler, A., Pardon, G., et al., “Transactions in Stack, Fork, and Join Composite Systems”. In: *Proceedings of the 7th International Conference on Database Theory (ICDT '99)*, Jerusalem, Israel, pp. 150-168, January 1999.

[4] Alonso, G., Fiedler, U., Hagen, C., et al., “WISE: Business to Business E-Commerce”. In: *Proceedings of the International Workshop on Research Issues in Data Engineering (RIDE)*, Sydney, Australia, 1999.

[5] Alonso, G., Schuldt, H., Schek, H., "Concurrency Control and Recovery in Transactional Process Management". In: *Proceedings of the Symposium on Principles of Database Systems in Philadelphia*, pp. 316-26, 1999.

[6] Andrade, J. M. , Carges, M. T. , MacBlane, M. R., “The TUXEDO System: An Open On-line Transaction Processing Environment”. *Data Engineering Bulletin*, v. 17, n. 1, pp. 34-39, 1994.

[7] Ansari, M., Ness, L., Rusinkiewicz, M., et al., “Using Flexible Transactions to Support Multi-System Telecommunication Applications”. In: *Proceedings of the 18th VLDB Conference*, August 1992.

[8] Arkin, A., "Business Process Modeling Language (BPML)”. [http://www.bpmi.org/bpml-spec.esp], BPMI.org, March 2001.

[9] Barbosa, V. C., Benevides, M. R. F., Oliveira, A. L. “Priority Dynamic for Generalized Drinking Philosophers”. *Information Processing Letters*, v. 79, pp. 189-195, 2001.

[10] BEA Systems Press Release, “Web Services Architecture of BEA WebLogic E-Business Platform Enables Real Business-to-Business Transactions and Collaboration over the Internet”. [http://www.bea.com/press/releases /2001/0226_web_services.shtml], February 2001.

[11] BEA, “BEA WebLogic Server, Programming WebLogic Enterprise JavaBeans”. BEA Systems Inc., March 2001.

[12] Berners-Lee, T., Fielding, R., Masinter, L., "Uniform Resource Identifiers (URI): Generic Syntax”. IETF RFC 2396, [http://ietf.org/rfc/ rfc2396.txt], August 1998.

[13] Bernstein, P. A., Goodman, N., “Concurrency Control in Distributed Database Systems”, *ACM Computing Surveys*, v. 13, n. 2, pp. 185-221, 1981.

[14] Bernstein, P. A., Hadizlacos, V. , Goodman, N., *Concurrency Control and Recovery in Databases Systems*. Addison-Wesley, USA, 1987.

[15] Bernstein, P. A., Newcomer, E., *Principles of Transaction Processing for Systems Professionals*. Morgan Kaufmann, ISBN 1-55860-415-4, 1996.

[16] Bourret, R., "XML Database Products". [http://www.rpbourret.com/xml/XMLDatabaseProds.htm], March 2002.

[17] Buchmann, A. P., Özsu, M. T., Georgakopoulos, D., et. al., "A Transaction Model for Active Distributed Object Systems". *Database Transaction Models for Advanced Applications*, Morgan Kaufmann, ISBN 1-55860-214-3, pp. 123-158, 1992.

[18] Buyya, R., Baker, M. (Eds.), *Grid Computing - GRID 2000, First IEEE/ACM International Workshop, Bangalore, India, December 17, 2000, Proceedings*. Lecture Notes in Computer Science, v. 1971, Springer Verlag, ISBN 3-540-41403-7, 2000.

[19] Carey, M. J., Haas, L. M., Schwarz, P. M., et al., *Towards Heterogeneous Multimedia Information Systems: The Garlic Approach*. Technical Report RJ 9911, IBM Almaden Research Center, USA, 1995.

[20] Carey, M. J., Mattos, N. M., Nori, A. K., "Object-relational database systems (tutorial): principles, products and challenges". In: *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, Tucson, Arizona, USA, 1997.

[21] Casanova, M. A., "The Concurrency Control Problem for Database Systems", *Lecture Notes in Computer Science*, v. 116, ISBN 3-540-10845-9, Springer 1981.

[22] Casati, F., Ilnicki, S., Jin, L., et al., "Adaptive and Dynamic Service Composition in eFlow". In: *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CaiSE 2000)*, Stockholm, Sweden, pp. 13-31, June 2000.

[23] Casati, F., Shan, M., "Dynamic and adaptive composition of e-services", *Information Systems*, v. 26, n. 3, pp. 143-163, 2001.

[24] Cattel, R. G. G., Barry, D., *The Object Databases Standard 2.0*. Morgan Kaufmann, USA, 1997.

[25] Champion, M., "Storing XML in Databases", *eAI journal*, [http://www.eaijournal.com/PDF/StoringXMLChampion.pdf], pp. 53-55, October 2001.

[26] Chen, J., Bukhres, O. A., and Elmagarmid, A. K., "IPL: A Multidatabase Transaction Specification Language". In: *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, PA, May 1993.

[27] Christophides, V., Hull, R., Karvounarakis, G., "Beyond Discrete E-Services: Composing Session-Oriented Services in Telecommunications". In: *Proceedings of the Second VLDB Workshop on Technologies for E-Services (TES 2001)*, Rome, Italy, pp. 58-73, September 2001.

[28] Codd, E. F., "A relational model of data for large shared data banks", *Communications of the ACM*, v. 13, n. 6, pp. 377-387, June 1970.

[29] Dayal, U., Hsu, M. and Ladin, R., "A Transactional Model for Long-Running Activities". In: *Proceedings of the 17th VLDB Conference*, September 1991.

[30] Dayal, U., Hsu, M., Ladin, R., “Organizing Long-Running Activities with Triggers and Transactions”. In: *Proceedings of ACM SIGMOD Conf. on Management of Data*, 1990.

[31] Elmagarmid, A. K., *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[32] Elmagarmid, A. K., Pu, C., eds., “Special Issue on Heterogeneous Databases”, *ACM Comp. Surveys* v. 22, n. 3, 1990.

[33] Elmagarmid, A., Leu, Y., Litwin, W., et al., "A Multidatabase Transaction Model for InterBase". In: *Proceedings of the 16thVLDB Conference*, Brisbane, Australia, pp. 507-18, 1990.

[34] Eswaran, K. P., Gray, J., Lorie, R. A., et al., “The Notions of Consistency and Predicate Locks in a Database System”, Communications of the ACM, v. 19, n.11, pp. 624-633, 1976.

[35] Evans, K., Klein, J. , Lyon, J., "Transaction Internet Protocol - Requirements and Supplemental Information”. IETF RFC 2372, [http://ietf.org/rfc/rfc2371.txt], 1998.

[36] Evans, K., Klein, J. , Lyon, J., "Transaction Internet Protocol Version 3.0”. IETF RFC 2372, [http://ietf.org/rfc/rfc2371.txt], 1998.

[37] Fielding, R., Gettys, J., Mogul, J., et al., "Hypertext Transfer Protocol - HTTP/1.1”. IETF RFC 2616, [http://ietf.org/rfc/rfc2616.txt], 2000.

[38] Freed, N., Borenstein N., “Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies”. IETF RFC 2045, [http://ietf.org/rfc/ rfc2045.txt], November 1996.

[39] Frolund, S., Govindarajan, K., “Transactional Conversations”. In: *W3C Web services Workshop*, [http://www.w3.org/2001/03/WSWS-popa/paper50], San Jose, CA, April 2001.

[40] Garcia-Molina, H., Gawlick, D., Klein, J., et al., “Modeling Long-Running Activities as Nested Sagas”, *Data Engineering Bulletin*, v. 14, n. 1, March 1991.

[41] Garcia-Molina, H., Gawlick, D., Klein, J., et al., *Coordinating Multi-transaction Activities*. Technical Report CS-TR-247-90, Princeton University, February 1990.

[42] Garcia-Molina, H., Papakonstantinou, Y., Quass, D., et al., “The TSIMMIS Approach to Mediation: Data Models and Languages”, *Journal of Intelligent Information Systems (JIIS)*, v. 8, n. 2, pp. 117-132, 1997.

[43] Garcia-Molina, H., Salem, K., “SAGAS”. In: *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1987.

[44] Georgakopoulos, D., Hornick, M. F., Krychniak, P., et al., “Specification and Management of Extended Transactions in a Programmable Transaction Environment”. In: *Proceedings of the Intl. Conf. on Data Engineering*, February 1994.

[45] Georgakopoulos, D., Hornick, M., Sheth, A., “An overview of workflow management: from process modeling to workflow automation infrastructure”, *Intl. Journal on distributed and parallel databases*, v. 3, n. 2 , pp. 119-153, 1995.

[46] Gray, J., “Notes on database Operating Systems”. In: *Operating Systems: An Advanced Course*, v. 60, Lecture Notes in Computer Science, Spring-Verlag, New York, 1978.

[47] Gray, J., Invited Paper, "The Transaction Concept: Virtues and Limitations". In: *Proceedings of the 7th International Conference on Very Large Data Bases*, pp. 144-154, September 1981.

[48] GRID Infoware, “Grid Computing Info Centre (GRID Infoware) - Home Page”. [http://www.gridcomputing.com/], 2002.

[49] Gruhn, V., “Business Process Modeling and Workflow Management”, *International Journal of Cooperative Information Systems IJCIS*, v. 4, n. 2-3, pp. 145-164, 1995.

[50] Gruser, J-R., Raschid, L., Vidal, M. E., et. al, “Wrapper Generation for Web Accessible Data Sources”. *In: Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems (CoopIS)*, New York City, New York, USA, pp. 14-23, August 1998.

[51] Haas, L., Kossmann, D., Wimmers, E. L., et al., “Optimizing Queries across Diverse Data Sources”. In: *Proceedings of the VLDB Conference*, 1997.

[52] Hadzilacos, V., “A theory of reliability in database systems”, *Journal of the ACM*, v. 35, n. 1, pp. 121-145, 1988.

[53] Halteren, A., Fábián, G., Groeneveld, E., “Design and Evaluation of a QoS Provisioning Service”. In: *Proceedings of the WG6.1 Third International Working Conference on Distributed Applications and Interoperable Systems (DAIS 2001)*, Kraków, Poland, pp. 189-202, September 2001.

[54] Härder, T. , Reuter, A., "Principles of Transaction-Oriented Database Recovery”, *ACM Computing Surveys*, v. 15, n. 4, pp. 287-317, December 1983.

[55] Heuvel, W., Yang, J., Papazoglou, M. P., “Service Representation, Discovery, and Composition for E-marketplaces”. In: *Proceedings of the 9th International Conference Cooperative Information Systems (CoopIS 2001)*, Trento, Italy, pp. 270-284, September 2001.

[56] Hsu, M., (ed.), *Special Issue on workflow and Extended Transaction Systems*. Bulletin of the Technical Committee on Data Engineering, v. 16, n.2, June 1993.

[57] Hyman, G., “IBM to Push Grid-Based Web Services ”. [http://www.internetnews .com/ent-news/article/0,,7_977291,00.html], *InternetNews eletronic magazine*, April 2002.

[58] IBM White Paper, “The IBM WebSphere software platform and patterns for e-business - invaluable tools for IT architects of the new economy”. [http://www4.ibm.com/software/info/websphere/ docs/wswhitepaper.pdf], 2000.

[59] IBM White Paper, “Web services by IBM”. [http://www-3.ibm.com/software /solutions/webservices/overview.html], 2002.

[60] IBM White Paper, “Web services Toolkit”. [http://www.alphaworks.ibm.com/ tech/webservicestoolkit], April 2002.

[61] IONA White paper, “IONA's Orbix E2A Web Services Integration Platform”. [http://www.iona.com/products/webserv.htm], 2002.

[62] IONA, *Orbix TM - A Technical Overview*. Technical Report PN: PR-TEC-7-5, IONA Technologies Ltd. Dublin, Ireland, 1993.

[63] Jacobson, I., Invited Paper, "The Unified Process for Component-Based Development". In: *Proceedings of the 11th International Conference on Advanced Information Systems Engineering (CAiSE'99)*, Heidelberg, Germany, June 14-18, 1999.

[64] Jajodia, S., Kerschberg, L. (eds), *Advanced Transaction Models and Architectures*. Kluwer, ISBN 0-7923-9880-7, 1997.

[65] Jennings, N. R., Norman, T. J., Faratin, P., "ADEPT: An Agent-Based Approach to Business Process Management", *SIGMOD Record*, v. 27, n. 4, pp. 32-39, 1998.

[66] Karp, A. H, Smathers, K., "Advertising and Discovering Business Services". In: *W3C Web services Workshop*, [http://www.w3.org/2001/03/WSWS-popa/paper09], San Jose, CA, April 2001.

[67] Kim, W., *Modern Database System: The Object Model, Interoperability, and Beyond*. ACM Press and Addison-Wesley, 1995.

[68] Korth, H. F., Levy, E., Silberschatz, A., "A Formal Approach to Recovery by Compensating Transactions". In: *Proceedings of the 16th International Conference on VLDB*, 1990.

[69] Krychniak, P., Rusinkiewicz, M., Cichocki, A., et al., "Bounding the Effects of Compensation under Relaxed Multi-level Serializability", *Distributed and Parallel Databases*, v. 4, n. 4, 1996.

[70] Lampson, B. W. , Lomet, D. B., "A New Presumed Commit Optimization for Two Phase Commit". In: *Proceedings of the 19th International Conference on VLDB*, pp. 630-640, 1993.

[71] Lampson, B. W., "Atomic Transactions". In: Goos, G., Hartmanis, J. (eds.), *Distributed Systems - Architecture and Implementation: An Advanced course*, Spring-Verlag, pp. 246-265, 1981.

[72] Lampson, B. W., Sturgis, H., *Crash Recovery in a Distributed Data Storage System.* Technical Report, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, CA, 1976.

[73] Lawrence, P., ed., *WfMC Workflow Handbook*. John Wiley & Sons Ltd., 1997.

[74] Lee, C. A. (Ed.), *Grid Computing - GRID 2001, Second International Workshop, Denver, CO, USA, November 12, 2001, Proceedings*. Lecture Notes in Computer Science, v. 2242, ISBN 3-540-42949-2, Springer 2001.

[75] Levy, A. Y., Rajaraman, A., Ordille, J. J., et al., "Querying Heterogeneous Information Sources Using Source Descriptions". In: *Proceedings of the VLDB Conference*, 1996.

[76] Leymann, F., "Web Services Flow Language (WSFL 1.0)". [http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf], May 2001.

[77] Liu, L., Pu, C., Lee, Y., "An Adaptive approach to query mediation across heterogeneous databases". In: *Proceedings of the Int. Conf. on Cooperative Information Systems*, IEEE Press, pp. 144-156, June 1996.

[78] Lynch, N. A., Merritt, M., Weihl, W. E., et al., *Atomic Transactions*. Morgan Kaufmann, ISBN 1-55860-104-X, 1993.

[79] McCarthy, D., Sarin, S., "Workflow and Transaction in InConcert". In [56].

[80] Mecella, M., Pernici, B., "Designing Wrapper Components for E-Services in Integrating Heterogeneous Systems", *The VLDB Journal*, v. 10, n. 1, pp. 2-15, 2001.

[81] Mecella, M., Pernici, B., "Designing Components for e-Services". In: *Proceedings of the First VLDB Workshop on Technologies for E-Services (TES 2000)*, Rome, Italy, pp. 58-73, September 2001.

[82] Medina-Mora, R., Wong, H. and Flores, P., "ActionWorkflow TM as the Enterprise Integration Technology". In [56].

[83] Meyer, B., Mingins, C., "Component-Based Development: From Buzz to Spark - Guest Editors' Introduction", *IEEE Computer*, v. 32, n. 7, pp. 35-37, 1999.

[84] Microsoft Corporation and Digital Equipment Corporation, "The Component Object Model Specification". [http://www.opengroup.org/pubs/catalog/ax01.htm], October 1995.

[85] Microsoft Corporation, ".Net Framework - Home Page". [http://msdn.microsoft.com /netframework/default.asp], 2000.

[86] Microsoft Corporation, "Guide to Microsoft Distributed Transaction Coordinator". In: *Microsoft SQL Server 6.5 documentation*, Redmond, WA, 1996.

[87] Microsoft Corporation, "Microsoft Distributed Transaction Coordinator - Home Page". [http://msdn.microsoft.com/library/en-us/cossdk/htm/pgdtcintro_05ki.asp], 2002.

[88] Microsoft White Paper, "A Blueprint for Building Web Sites Using the Microsoft Windows DNA Platform". [http://www.microsoft.com/commerceserver/techres/ whitepapers.asp], 2000.

[89] Microsoft White Paper, "A Platform for Web Services". [http://msdn.microsoft.com /library/en-us/dndotnet/html/Techmap_websvcs.asp], 2001.

[90] Miller, J. A, Palaniswami, D., Sheth, A. P., et al., "WebWork: METEOR$_2$'s Web-Based Workflow Management System", *Journal of Intelligent Information Systems*, v.10, n. 2, pp. 185-215, 1998.

[91] Mohan, C., "Transaction Processing and Distributed Computing in the Internet Age". In: *Proceeding of the International Conference on Extending Database Technology (EDBT)*, Tutorial Notes, Valencia, Spain, 1998.

[92] Mohan, C., Lindsay, B., Obermarck., R., "Transaction Management in the R* Distributed Database Management System", *ACM Transactions on Database Systems*, v. 11, n. 4, pp. 378-396, 1986.

[93] Moss, J. E. B., *Nested Transactions: An Approach to Reliable Distributed Computing*. Ph.D. thesis, MIT Press, Cambridge, MA, 1985.

[94] Navathe, S. B., Tanaka, A. K., Chakravarthy, S., "Active Database Modeling and Design Tools: Issues, Approache, and Architecture", *IEEE Data Engineering Bulletin*, v. 15, n. 1-4, pp. 6-9, 1992.

[95] Nielsen, H., Leach, P. , Lawrence, S., "An HTTP Extension Framework". IETF RFC 2774, [http://ietf.org/rfc/rfc2774.txt], 2000.

[96] OMG (Object Management Group). "Transaction Service Specificaiton". Revision 1.2.1, [http://www.omg.org/technology/documents/formal/transaction_service.htm], 2002.

[97] OMG (Object Management Group). The Common Object Request Broker: Architecture and Specification. Revision 2.0., July 1995.

[98] Özsu, M. T., Dayal, U., Valduriez, P., eds., *Distributed Object Management*. Morgan Kaufmann, San Mateo, CA, 1994.

[99] Papadimitriou, C. H., *The Theory of Database Concurrency Control*, Computer Science Press, 1986.

[100] Pardon, G., Alonso, G., "CheeTah: a Lightweight Transaction Server for Plug-and-Play Internet Data Management". In: *Proceedings of the 26th International Conference on Very Large Data Bases*, Cairo, Egypt, pp. 210-219, September 2000.

[101] Paul, S., Park, E., Chaar, J., "RainMan: a Workflow System for the Internet". In: *Proceedings of USENIX Symp. on Internet Technologies and Systems*, 1997.

[102] Phillips, J., Foody, D., "Building a Fundation for Web Services", *eAI Journal*, [http://www.eaijournal.com/PDF/WSFoundationFoody.pdf], pp. 8-13, March 2002.

[103] Pires, P. F., Benevides, M. R. F., Mattoso, M. L. Q., "Mechanisms for Specifying Communication Behavior in Object Oriented Database". In: *Proceedings of the 2000 ACM Symposium on Applied Computing*, Como, Italy, pp. 389-397, v. 1, March 2000.

[104] Pires, P. F., Brügger, T., Mattoso, M. L. Q., "Mediators Metadata Management Services : An Implementation Using GOA++ System", *Electronic Journal of SADIO*, [http://www.dc.uba.ar/sadio/ejs], v. 2, n. 1, pp. 30-47, 1999.

[105] Pires, P. F., Mattoso, M. L. Q., "A CORBA Based Architecture for Heterogeneous Information Source Interoperability". In: *Proceedings of the 25th Technology of Object-Oriented Languages and Systems (TOOLS-25 '97)*, November, 1997, Melbourne Australia. IEEE Press, ISBN 0-8186-8485-2, June 1998.

[106] Pires, P. F., Mattoso, M. L. Q., "Aspectos de Implementação na Arquitetura Heterogênea HIMPAR [Interoperability Issues in the HIMPAR Heterogeneous Architecture]," Proc. of the XXI Brazilian Symp. Databases, São Carlos, Brazil, pp. 43-57, 1996 (in Portuguese).

[107] Pires, P. F., Raschid, L., "MedTransact: Transaction Support for Mediation with Remote Service Providers". In: *Proceedings of the 3rd International Conference on Telecommunications and Electronic Commerce*, Dallas, USA, November, 2000.

[108] Pitoura, E., Bukhres, O. A. and Elmagarmid, A. K., "Object Orientation in Multidatabase Systems", *ACM Computing Surveys*, v. 27, n. 2, pp. 141-195, 1995.

[109] Pree, W., "Component-Based Software Development - A New Paradigm in Software Engineering?", *Software - Concepts and Tools*, v. 18, n. 4, pp. 169-174, 1997.

[110] Ramamritham, K., Chrysanthis, P. K., ed., "Advances in Concurrency Control and Transaction Processing", *IEEE Computer Society Press*, CA, 1997.

[111] Ranno, F., Shrivastava, S. K., Wheater, S. M., "A Language for Specifying the Composition of Reliable Distributed Applications". In: *Proceedings of the 18th Intl. Conf. on Distributed Computing Systems (ICDCS '98)*, Amsterdam, The Netherlands 1998.

[112] Rusinkiewicz, M., Sheth, A., "Specification and execution of transactional workflows". In: **[67]**, pp. 592-620.

[113] Schuldt, H., Alonso, G., Schek, H. J., "Concurrency Control and Recovery in Transactional Process Management". In: *Proceedings of the Symposium on Principles of Database Systems (PODS)*, Philadelphia, Pennsylvania, pp. 316-326, 1999.

[114] Schuldt, H., Schek, H. J., Alonso, G., "Transactional Coordination Agents for Composite Systems". In: *Proceedings of the International Database Engineering and Applications Symposium (IDEAS 1999)*, Montreal, Canada, pp. 321-331, August 1999.

[115] Seltzsam, S., Börzsönyi, S., Kemper, A., "Security for Distributed E-Service Composition". In: *Proceedings of the Second VLDB Workshop on Technologies for E-Services (TES 2001)*, Rome, Italy, pp. 147-162, September 2001.

[116] Shan, M., Davis, J., Du, W., et al., *HP Workflow Research: Past, Present, and Future*. Technical Report, Hewlett Packard Software Technology Laboratory, [http://www.hpl.hp.com/techreports/97/HPL-97-105.html], 1997.

[117] Sherman, M., "Architecture of the Encina Distributed Transaction Processing Family". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, D.C., USA, pp. 460-463, May 1993.

[118] Short, S., "Building XML Web Services for the Microsoft® .NET Platform". ISBN 0-7356-1406-7, Microsoft Press, February 2002.

[119] Shread, P., "IBM To Announce Broad Support For Grid Computing, Globus". [http://www.internetnews.com/ent-news/article/0,,7_977291,00.html], *InternetNews eletronic magazine*, February 2002.

[120] Smerik, R., "TOP END: Distributed Transaction Processing for Open Systems". In: *Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems (PDIS 1993)*, San Diego, CA, USA, pp. 270-271, January 1993.

[121] Stonebraker, M., "Object-Relational Database Systems", *White-paper Montage Software Inc. (Now Called Illustra)*, 1994.

[122] Sullivan, T., "Sun combines grid computing, Web services". [http://www.infoworld.com/articles/hn/xml/02/02/15/020215hnsunonegrid.xml], *InforWorld eletronic magazine*, Feburary 2002.

[123] SUN Microsystems, "Developing Web Services with the Sun Open Net Environment". [http://www.sun.com/software/sunone/wp-spine/spine.pdf], 2002.

[124] SUN Microsystems, "Enterprise JavaBeans Specification 2.0. Sun Microsystems". [http://java.sun.com/products/ejb/docs.html], August 2001.

[125] SUN Microsystems, "Implementing Services on Demand and the Sun Open Net Environment (Sun ONE)". [http://www.sun.com/software/sunone/wp-implement/wp-implement.pdf], 2001.

[126] SUN Microsystems, “Java Remote Method Invocation (RMI)”. [http://java.sun.com/ j2se/1.4/docs/guide/rmi/spec/rmiTOC.html], 2001.

[127] SUN Microsystems, “Java Transaction Service (JTS)”. [http://java.sun.com/ products/jts/], 2002.

[128] Tanaka, A. K., Valduriez, P., “The Ecobase Project: Database and Web Technologies for Environmental Information Systems”, *SIGMOD Record*, v. 30, n. 3, pp. 70-75, 2001.

[129] Thatte, S., “XLANG: Web Services for Business Process Design”. [http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm], Microsoft Corporation, 2001.

[130] The Open Group, “Distributed Transaction Processing: XA+ Specification”. [http://www.opengroup.org/dbiop/s423.pdf], Version 2, ISBN: 1-85912-046-6, June 1994.

[131] The Open Group, “X/Open Guide - Distributed Transaction Processing: Reference Model”. Version 3, 1996.

[132] Tomasic, A., Raschid, L., Valduriez, P., “Scaling Access to Heterogeneous Data Sources with DISCO”, *IEEE Transactions on Knowledge and Data Engineering*, v. 10, n. 5, pp. 808-823, 1998.

[133] UDDI.org, “UDDI Technical White Paper”. [http://www.uddi.org/pubs/ Iru_UDDI_Technical_White_Paper.PDF], September 2000.

[134] Vassalos, V., Papakonstantinou, Y., “Describing and Using Query Capabilities of Heterogeneous Sources”. In: *Proceedings of the VLDB Conference*, pp. 256-265, 1997.

[135] Vidal, M. E., Raschid, L., Gruser, J-R., “A Meta-Wrapper for Scaling up to Multiple Autonomous Distributed Information Sources”. *In: Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems (CoopIS)*, New York City, New York, USA, pp. 148-157, August 1998.

[136] W3C (World Wide Web Consortium) Note, "Web Services Conversation Language (WSCL) 1.0”. [http://www.w3.org/TR/2002/NOTE-wscl10-20020314/], March 2001.

[137] W3C (World Wide Web Consortium) Note, "Web Services Description Language (WSDL) 1.1”. [http://www.w3.org/TR/2001/NOTE-wsdl-20010315], March 2001.

[138] W3C (World Wide Web Consortium) Recommendation, "Extensible Markup Language (XML) 1.0 (Second Edition)”. [http://www.w3.org/TR/REC-xml], October 2000.

[139] W3C (World Wide Web Consortium) Recommendation, "HTML 4.01 Specification”. [http://www.w3.org/TR/1999/REC-html401-19991224], December 1999.

[140] W3C (World Wide Web Consortium) Recommendation, "XML Path Language”. [http://www.w3.org/TR/1999/REC-XPath-19991116], November 1999.

[141] W3C (World Wide Web Consortium) Recommendation, "XML Schema Part 1: Structures”. [http://www.w3.org/TR/xmlschema-1/], May 2001.

[142] W3C (World Wide Web Consortium) Recommendation, "XML Schema Part 2: Datatypes". [http://www.w3.org/TR/xmlschema-2/], May 2001.

[143] W3C (World Wide Web Consortium) Recommendation, "XML Schema Part 0: Primer". [http://www.w3.org/TR/xmlschema-0/], May 2001.

[144] W3C (World Wide Web Consortium) Working Draft, "SOAP Version 1.2 Part 0: Primer". [http://www.w3.org/TR/soap12-part0/], December 2001.

[145] Wachter H., Reuter A., "The ConTract Model". In: **[31]**, Chapter 7.

[146] Weikum, G., Schek, H. J., "Concepts and Applications of Multilevel Transactions and Open Nested Transactions". In: [31] , Chapter 13.

[147] Widya, I., Stap, R. E., Teunissen, L. J. , et al., "On the End-User QoS-Awareness of a Distributed Service Environment". In: *Proceedings of the 6th International Conference on Protocols for Multimedia Systems (PROMS 2001)*, Enschede, The Netherlands, pp. 222-238, October 2001.

[148] Wiederhold, G., "Mediation in Information Systems", *ACM Computing Surveys*, v. 27, n. 2, pp. 265-267, 1995.

[149] Yeong, W. , Howes, T. , Kille, S., "Lightweight Directory Access Protocol". IETF RFC 2372, [http://ietf.org/rfc/rfc1777.txt], 1995.

[150] Zadorozhny, V., Raschid, L., Vidal, M. E., et. al, "Efficient Evaluation of Queries in a Mediator for WebSources". In: *Eletronic Proceedings of the ACM SIGMOD Conference*, [http://www.acm.org/sigs/sigmod/sigmod02/eproceedings/papers/Research-Zadorozhny-et-al.pdf], 2002.

[151] Zhang, A., Nodine, M. H., Bhargava B. K., et al., "Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems". In: *Proceedings of the ACM SIGMOD Conference*, pp. 67-78, 1994.

Appendix 1 - The car reservation service

The car reservation service - WSDL document extended by WSTL framework.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions targetNamespace="http://example.com/carReservation/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:lxsd="http://example.com/carReservation/Schema/"
  xmlns:wstl="http://schemas.xmlsoap.org/wsdl/wstl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"

  <types>
    <s:schema attributeFormDefault="qualified"
              elementFormDefault="qualified"
              targetNamespace="http://example.com/carReservation/Schema/">
      <s:element name="reservation">
        <s:complexType>
          <s:sequence>
            <s:element name="pInput" type="s0:reservationInputType"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:complexType name="reservationInputType">
        <s:sequence>
          <s:element name="pickupInfo" type="s0:infoType"/>
          <s:element name="dropoffInfo" type="s0:infoType"/>
        </s:sequence>
      </s:complexType>
      <s:complexType name="infoType">
        <s:sequence>
          <s:element name="dt" type="s:dateTime"/>
          <s:element name="location" type="s:string"/>
          <s:element name="time" type="s:int"/>
        </s:sequence>
      </s:complexType>
      <s:element name="reservationResponse">
        <s:complexType>
          <s:sequence>
            <s:element name="reservationResult" type="s:string"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="cancelReservation">
        <s:complexType>
          <s:sequence>
            <s:element name="reservationCode" type="s:string"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="cancelReservationResponse">
        <s:complexType>
          <s:sequence>
            <s:element name="cancelReservationResult" type="s:string"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="string" type="s:string"/>
    </s:schema>
  </types>
  <message name="reservationSoapIn">
    <part name="parameters" element="s0:reservation"/>
  </message>
  <message name="reservationSoapOut">
    <part name="parameters" element="s0:reservationResponse"/>
  </message>
  <message name="cancelReservationSoapIn">
    <part name="parameters" element="s0:cancelReservation"/>
  </message>
  <message name="cancelReservationSoapOut">
    <part name="parameters" element="s0:cancelReservationResponse"/>
  </message>
  <portType name="reservationSoap">
    <operation name="reservation">
      <input message="s0:reservationSoapIn"/>
```

```
    <output message="s0:reservationSoapOut"/>
  </operation>
  <operation name="cancelReservation">
    <input message="s0:cancelReservationSoapIn"/>
    <output message="s0:cancelReservationSoapOut"/>
  </operation>
</portType>
<binding name="reservationSoap" type="s0:reservationSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
               style="document"/>
  <operation name="reservation">
    <soap:operation soapAction="http://example.com/reservation"
                style="document"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
  <operation name="cancelReservation">
    <soap:operation
        soapAction="http://example.com/cancelReservation" style="document"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
<service name="carReservation">
  <port name="reservationSoap" binding="s0:reservationSoap">
    <soap:address location="http://example.com/carReservation/"/>
  </port>
</service>

<wstl:transactionDefinitions>
  <wstl:transactionBehavior operationName="reservation" type="compensable">
    <wstl:activeAction portTypeName="reservationSoap"
                compensatoryOper="cancelReservation">
      <wstl:paramLink>
        <wstl:sourceParamLink msgName="reservationSoapOut"
               param="s0:reservationResponse/@reservationResult"/>
        <wstl:targetParamLink msgName="cancelReservationSoapIn"
               param="s0:cancelReservation/@reservationCode"/>
      </wstl:paramLink>
    </wstl:activeAction>
  </wstl:transactionBehavior>
  <wstl:transactionBehavior operationName="cancelReservation"
                  type="retriable"/>
</wstl:transactionDefinitions>
</definitions>
```

Appendix 2 - The Extended Car Reservation Service

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v4.1 U (http://www.xmlspy.com) by Flavia (HO) -->
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:wstl="http://schemas.xmlsoap.org/wsdl/wstl/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://schemas.xmlsoap.org/wsdl/
http://localhost/Schemas/wstl.xsd">
  <import namespace="http://schemas.xmlsoap.org/wsdl/wstl/"
location="http://localhost/Schemas/wstl.xsd"/>
  <types>
    <s:schema attributeFormDefault="qualified" elementFormDefault="qualified"
              targetNamespace="http://tempuri.org/">
      <s:element name="reservation">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="pInput"
                       type="s0:reservationInputType"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:complexType name="reservationInputType">
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" name="pickupInfo" type="s0:infoType"/>
          <s:element minOccurs="1" maxOccurs="1" name="dropoffInfo" type="s0:infoType"/>
        </s:sequence>
      </s:complexType>
      <s:complexType name="infoType">
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" name="dt" type="s:dateTime"/>
          <s:element minOccurs="1" maxOccurs="1" name="location" nillable="true"
                     type="s:string"/>
          <s:element minOccurs="1" maxOccurs="1" name="time" type="s:int"/>
        </s:sequence>
      </s:complexType>
      <s:element name="reservationResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="reservationResult" nillable="true"
                       type="s:string"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="cancelReservation">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="reservationCode" nillable="true"
                       type="s:string"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="cancelReservationResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="cancelReservationResult"
                       nillable="true" type="s:string"/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="string" nillable="true" type="s:string"/>
    </s:schema>
  </types>
  <message name="reservationSoapIn">
    <part name="parameters" element="s0:reservation">
      <documentation/>
    </part>
  </message>
  <message name="reservationSoapOut">
    <part name="parameters" element="s0:reservationResponse">
      <documentation/>
    </part>
  </message>
```

```
<message name="cancelReservationSoapIn">
  <part name="parameters" element="s0:cancelReservation">
    <documentation/>
  </part>
</message>
<message name="cancelReservationSoapOut">
  <part name="parameters" element="s0:cancelReservationResponse">
    <documentation/>
  </part>
</message>
<message name="cancelReservationHttpGetIn">
  <part name="reservationCode" type="s:string">
    <documentation/>
  </part>
</message>
<message name="cancelReservationHttpGetOut">
  <part name="Body" element="s0:string">
    <documentation/>
  </part>
</message>
<message name="cancelReservationHttpPostIn">
  <part name="reservationCode" type="s:string">
    <documentation/>
  </part>
</message>
<message name="cancelReservationHttpPostOut">
  <part name="Body" element="s0:string">
    <documentation/>
  </part>
</message>
<portType name="reservationSoap">
  <operation name="reservation">
    <input message="s0:reservationSoapIn"/>
    <output message="s0:reservationSoapOut"/>
  </operation>
  <operation name="cancelReservation">
    <input message="s0:cancelReservationSoapIn"/>
    <output message="s0:cancelReservationSoapOut"/>
  </operation>
</portType>
<portType name="reservationHttpGet">
  <operation name="cancelReservation">
    <input message="s0:cancelReservationHttpGetIn"/>
    <output message="s0:cancelReservationHttpGetOut"/>
  </operation>
</portType>
<portType name="reservationHttpPost">
  <operation name="cancelReservation">
    <input message="s0:cancelReservationHttpPostIn"/>
    <output message="s0:cancelReservationHttpPostOut"/>
  </operation>
</portType>
<wstl:trasactionDefinitions>
  <wstl:transactionBehavior operationName="s0:reservation" type="compensable">
    <documentation/>
    <wstl:activeAction portTypeName="s0:reservationSoap"
                       compensatoryOper="s0:cancelReservation">
      <documentation/>
      <wstl:msgParamLink>
        <wstl:sourceParamLink msgName="s0:reservationSoapOut"
                      param="s0:reservationResponse/@reservationResult"/>
        <wstl:targetParamLink msgName="s0:cancelReservationSoapIn"
                      param="s0:cancelReservation/@reservationCode"/>
      </wstl:msgParamLink>
    </wstl:activeAction>
    <wstl:activeAction portTypeName="s0:reservationHttpGet"
                       compensatoryOper="s0:cancelReservation">
      <documentation/>
      <wstl:msgParamLink>
        <wstl:sourceParamLink msgName="s0:reservationSoapOut"
                      param="s0:reservationResponse/@reservationResult"/>
        <wstl:targetParamLink msgName="s0:cancelReservationHttpGetIn"
                      param="@reservationCode"/>
      </wstl:msgParamLink>
    </wstl:activeAction>
    <wstl:activeAction portTypeName="s0:reservationHttpPost"
                       compensatoryOper="s0:cancelReservation">
```

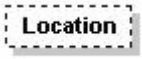
```
      <documentation/>
      <wstl:msgParamLink>
        <wstl:sourceParamLink msgName="s0:reservationSoapOut"
                      param="s0:reservationResponse/@reservationResult"/>
        <wstl:targetParamLink msgName="s0:cancelReservationHttpPostIn"
                      param="@reservationCode"/>
      </wstl:msgParamLink>
    </wstl:activeAction>
  </wstl:transactionBehavior>
  <wstl:transactionBehavior operationName="cancelReservation" type="retriable"/>
 </wstl:trasactionDefinitions>
</definitions>
```

Appendix 3 - Graphical Notation for XML Schema elements

Element Symbols:

Mandatory single element. Details: MinOcc=1, MaxOcc=1

Mandatory single element, containing Parsed Character Data (#PC-Data). The content may be simple content or mixed complex content. Details: MinOcc=1, MaxOcc=1, type=xsd:string, content=simple.



Single optional element. Details: MinOcc=0, MaxOcc=1.

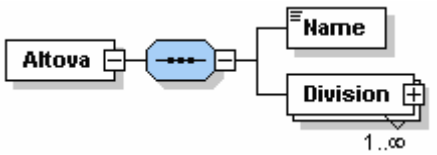
Mandatory multiple element. Details MinOcc=1, MaxOcc=5.

Mandatory multiple element containing child elements. Details: MinOcc=1, MaxOcc=unbounded, type=DivisionType, content=complex.

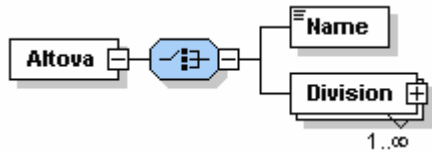
Group element. Details: name=Subsidiaries. A named collection of elements to allow reuse in the construction of different complex types.

Compositors:

A "Compositor" defines an ordered sequence of sub elements (child elements).



Sequence.



Choice.

Appendix 4 - WSTL Schema

XSD Schema for the WSTL elements that define the transaction behavior of Web Services operations.

```
<xsd:schema targetNamespace="http://schemas.xmlsoap.org/wsdl/wstl/"
  xmlns:wstl="http://schemas.xmlsoap.org/wsdl/wstl/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tmmap="http://schemas.xmlsoap.org/wsdl/tmMap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
  attributeFormDefault="unqualified">

  <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/tmMap/"
              schemaLocation="http://localhost/Schemas/tmmap.xsd"/>
  <xsd:element name="transactionDefinitions" type="wstl:transactionDefinitionsType">
    <xsd:keyref name="compensatoryOperKey" refer="wsdl:operation">
      <xsd:selector XPath="wstl:transactionBehavior/wstl:activeAction"/>
      <xsd:field XPath="@compensatoryOper"/>
    </xsd:keyref>
    <xsd:keyref name="portTypeNameKey" refer="wsdl:portType">
      <xsd:selector XPath="wstl:transactionBehavior/wstl:activeAction"/>
      <xsd:field XPath="@portTypeName"/>
    </xsd:keyref>
    <xsd:keyref name="sourceParamLinkKeyRef" refer="wsdl:message">
      <xsd:selector XPath="wstl:transactionBehavior/
                           wstl:activeAction/wstl:paramLink/wstl:sourceParamLink"/>
      <xsd:field XPath="@msgName"/>
    </xsd:keyref>
    <xsd:keyref name="targetParamLinkKeyRef" refer="wsdl:message">
      <xsd:selector XPath="wstl:transactionBehavior/
                           wstl:activeAction/wstl:paramLink/wstl:targetParamLink"/>
      <xsd:field XPath="@msgName"/>
    </xsd:keyref>
    <xsd:keyref name="tmPortKeyRef" refer="wsdl:port">
      <xsd:selector XPath="wstl:transactionBehavior/wstl:tmSrv"/>
      <xsd:field XPath="@tmPort"/>
    </xsd:keyref>
    <xsd:keyref name="tmMapKeyRef" refer="tmmap:tmSrvMapKey">
      <xsd:selector XPath="wstl:transactionBehavior/wstl:tmSrv"/>
      <xsd:field XPath="@tmMap"/>
    </xsd:keyref>
    <xsd:unique name="tmSrvTmElemUnique">
      <xsd:selector XPath="wstl:transactionBehavior"/>
      <xsd:field XPath="@operationName"/>
      <xsd:field XPath="wstl:tmSrv/wstl:tmElem/@tmPort"/>
      <xsd:field XPath="wstl:tmSrv/wstl:tmElem/@tmMap"/>
    </xsd:unique>
    <xsd:unique name="tmSrvTmRefUnique">
      <xsd:selector XPath="wstl:transactionBehavior"/>
      <xsd:field XPath="@operationName"/>
      <xsd:field XPath="wstl:tmSrv/wstl:tmRef/@tmElemName"/>
    </xsd:unique>
    <xsd:keyref name="tmSrvtmElemKeyRef" refer="wstl:tmElemKey">
      <xsd:selector XPath="wstl:transactionBehavior/wstl:tmSrv/wstl:tmRef"/>
      <xsd:field XPath="@tmElemName"/>
    </xsd:keyref>
    <xsd:key name="tmElemKey">
      <xsd:selector XPath="wstl:tmElem"/>
      <xsd:field XPath="@name"/>
    </xsd:key>
    <xsd:key name="transactionBehaviorKey">
      <xsd:selector XPath="wstl:transactionBehavior"/>
      <xsd:field XPath="@operationName"/>
    </xsd:key>
    <!--  <xsd:keyref name="transactionBehaviorOperKeyRef" refer="wsdl:operation">
      <xsd:selector XPath="wstl:transactionBehavior"/>
      <xsd:field XPath="@operationName"/>
    </xsd:keyref> -->
  </xsd:element>
  <xsd:complexType name="transactionDefinitionsType">
    <xsd:complexContent>
      <xsd:extension base="wsdl:documented">
        <xsd:sequence minOccurs="0">
```

```
      <xsd:element ref="wstl:transactionBehavior" minOccurs="0" maxOccurs="unbounded"/>
      <xsd:element ref="wstl:tmElem" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="tmElem" type="wstl:tmElemType"/>
<xsd:complexType name="tmElemType">
 <xsd:complexContent>
  <xsd:extension base="wsdl:documented">
   <xsd:attribute name="name" type="xsd:NCName" use="optional"/>
   <xsd:attribute name="tmPort" type="xsd:QName" use="required"/>
   <xsd:attribute name="tmMap" type="xsd:QName" use="required"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="transactionBehavior" type="wstl:transactionBehaviorType"/>
<xsd:complexType name="transactionBehaviorType">
 <xsd:complexContent>
  <xsd:extension base="wsdl:documented">
   <xsd:choice minOccurs="0">
    <xsd:group ref="wstl:compensable"/>
    <xsd:group ref="wstl:virtualCompensable"/>
   </xsd:choice>
   <xsd:attribute name="name" type="xsd:NMTOKEN" use="optional"/>
   <xsd:attribute name="type" type="wstl:transactionBehaviorEnum" use="required"/>
   <xsd:attribute name="operationName" type="xsd:QName" use="required"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:simpleType name="transactionBehaviorEnum">
 <xsd:restriction base="xsd:string">
  <xsd:enumeration value="pivot"/>
  <xsd:enumeration value="compensable"/>
  <xsd:enumeration value="virtualCompensable"/>
  <xsd:enumeration value="retriable"/>
 </xsd:restriction>
</xsd:simpleType>
<xsd:group name="virtualCompensable">
 <xsd:sequence>
  <xsd:element ref="wstl:tmSrv"/>
 </xsd:sequence>
</xsd:group>
<xsd:group name="compensable">
 <xsd:choice>
  <xsd:element ref="wstl:activeAction" maxOccurs="unbounded"/>
  <xsd:element ref="wstl:passiveAction"/>
 </xsd:choice>
</xsd:group>
<xsd:element name="tmSrv" type="wstl:tmSrvType"/>
<xsd:complexType name="tmSrvType">
 <xsd:choice>
  <xsd:element ref="wstl:tmRef" maxOccurs="unbounded"/>
  <xsd:element ref="wstl:tmElem" maxOccurs="unbounded"/>
 </xsd:choice>
</xsd:complexType>
<xsd:element name="tmRef" type="wstl:tmRefType"/>
<xsd:complexType name="tmRefType">
 <xsd:complexContent>
  <xsd:extension base="wsdl:documented">
   <xsd:attribute name="tmElemName" type="xsd:QName" use="required"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
<xsd:element name="activeAction" type="wstl:activeActionType"/>
<xsd:complexType name="activeActionType">
 <xsd:complexContent>
  <xsd:extension base="wsdl:documented">
   <xsd:sequence>
    <xsd:element ref="wstl:msgParamLink" minOccurs="0" maxOccurs="unbounded"/>

   </xsd:sequence>
   <xsd:attribute name="name" type="xsd:NMTOKEN" use="optional"/>
   <xsd:attribute name="portTypeName" type="xsd:QName" use="required"/>
   <xsd:attribute name="compensatoryOper" type="xsd:QName" use="required"/>
  </xsd:extension>
 </xsd:complexContent>
```

```
  </xsd:complexType>
  <xsd:element name="msgParamLink" type="wstl:msgParamLinkType"/>
  <xsd:complexType name="msgParamLinkType">
    <xsd:complexContent>
      <xsd:extension base="wsdl:documented">
        <xsd:sequence>
          <xsd:element name="sourceParamLink" type="wstl:msgLinkType"/>
          <xsd:element name="targetParamLink" type="wstl:msgLinkType"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:NMTOKEN" use="optional"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:complexType name="msgLinkType">
    <xsd:complexContent>
      <xsd:extension base="wsdl:documented">
        <xsd:attribute name="name" type="xsd:NMTOKEN" use="optional"/>
        <xsd:attribute name="msgName" type="xsd:QName" use="required"/>
        <xsd:attribute name="param" type="tmmap:XPathLinkType" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="passiveAction" type="wstl:passiveActionType"/>
  <xsd:complexType name="passiveActionType">
    <xsd:complexContent>
      <xsd:extension base="wsdl:documented">
        <xsd:attribute name="name" type="xsd:NMTOKEN" use="optional"/>
        <xsd:attribute name="expiration" type="xsd:double" use="required"/>
        <xsd:attribute name="unit" type="wstl:timeUnitType" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="timeUnitType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="minutes"/>
      <xsd:enumeration value="hours"/>
      <xsd:enumeration value="days"/>
      <xsd:enumeration value="weeks"/>
      <xsd:enumeration value="moths"/>
      <xsd:enumeration value="year"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Appendix 5 - TIP Commands Interface.

This Appendix contains the description of the standard transaction manager to transaction manager interface of the WSTL framework. Basically, this interface is the set of TIP commands described in [36], the only difference is the addition of the connection identifier. In TIP, the transaction managers that share transactions must establish a TCP connection. In the WSTL framework, the transaction manager itself is a Web service. Therefore, it is not possible to guarantee the transport protocol to be TCP. To provide an equivalent semantics to TCP connections, the WSTL framework enforces transaction mangers to use connection identifiers when sending TIP commands. This guarantees the TCP connection semantics even when a connectionless protocol is been used. The connection identifier generation follows the same rules of the TIP transaction identifiers as defined in the Section "TIP Uniform Resource Locators" in [36].

TIP commands relate either to connections or transactions. The following commands are relate to connections: IDENTIFY, MULTIPLEX and TLS. Commands that relate to transactions are: ABORT, BEGIN, COMMIT, PREPARE, PULL, PUSH, QUERY, and RECONNECT. A list of all commands supported by the WSTL framework is presented in the following paragraphs. The MULTIPLEX and TLS are currently not supported.

1) ABORT. This command is valid in the Begun, Enlisted, and Prepared states. It informs the secondary that the current transaction of the connection will abort.

   a) Input parameters:

      i) <connectionId> - connection identifier.

   b) Possible Responses:

      i) [ABORTED] - the transaction has aborted; the connection enters Idle state.

      ii) [ERROR] - the command was issued in the wrong state, or was malformed. The connection enters the Error state.

2) BEGIN. This command is valid only in the Idle state. It asks the secondary to create a new transaction and associate it with the connection. The newly created transaction will be completed with a one-phase protocol.

   a) Input parameters:

      i) <connectionId> - connection identifier.

   b) Possible Responses:

      i) <transaction identifier> - a new transaction has been successfully begun, and that transaction is now the current transaction of the connection. The connection enters Begun state.

      ii) [NOTBEGUN] - a new transaction could not be begun; the connection remains in Idle state.

      iii) [ERROR] - The command was issued in the wrong state, or was malformed. The connection enters the Error state.

3) COMMIT. This command is valid in the Begun, Enlisted or Prepared states. In the Begun or Enlisted state, it asks the secondary to attempt to commit the transaction; in the Prepared state, it informs the secondary that the transaction has committed. Note that in the Enlisted state this command represents a one-phase protocol, and should only be done when the sender has 1) no local recoverable resources involved in the transaction, and 2) only one subordinate (the sender will not be involved in any transaction recovery process).

   a) Input parameters:

      i) <connectionId> - connection identifier.

   b) Possible Responses:

      i) [ABORTED] - this response is possible only from the Begun and Enlisted states. It indicates that some party has vetoed the commitment of the transaction, so it has been aborted instead of committing. The connection enters the Idle state.

ii) [COMMITTED] - this response indicates that the transaction has been committed, and that the primary no longer has any responsibilities to the secondary with respect to the transaction. The connection enters the Idle state.

iii) [ERROR] - the command was issued in the wrong state, or was malformed. The connection enters the Error state.

4) ERROR. This command is valid in any state; it informs the secondary that a previous response was not recognized or was badly formed. A secondary should not respond to this command. The connection enters Error state.

5) IDENTIFY. This command is valid only in the Initial state. The primary party generates a new connection identifier and informs the secondary party of: 1) The connection identifier; 2) the lowest and highest protocol version supported (all versions between the lowest and highest must be supported; 3) optionally, an identifier for the primary party at which the secondary party can re-establish a connection if ever needed (the primary transaction manager address); and 4) an identifier which may be used by intermediate proxy servers to connect to the required TIP transaction manager (the secondary transaction manager address). If a primary transaction manager address is not supplied, the secondary party will respond with ABORTED or READONLY to any PREPARE commands.

a) Input parameters:

i) <connectionId> - connection identifier

ii) <lowestProtocolVersion>

iii) <highestProtocolVersion>

iv) <primaryTMUrl> - primary transaction manager address

v) <secundaryTMUrl> - secondary transaction manager address

b) Possible responses:

i) <protocolVersion> - the secondary party has been successfully contacted and has saved the primary transaction manager address. The response contains the highest protocol version supported by the secondary party. All future communication is assumed to take place using the smaller of the protocol versions in the IDENTIFY command and the IDENTIFIED response. The connection enters the Idle state.

ii) [ERROR] - the command was issued in the wrong state, or was malformed. This response also occurs if the secondary party does not support any version of the protocol in the range supported by the primary party. The connection enters the Error state. The primary party should close the connection.

6) PREPARE. This command is valid only in the Enlisted state; it requests the secondary to prepare the transaction for commitment (phase one of two-phase commit).

a) Input parameters:

i) <connectionId>- connection identifier

b) Possible responses:

c) [PREPARED] - the subordinate has prepared the transaction; the connection enters PREPARED state.

d) [ABORTED] - the subordinate has vetoed committing the transaction. The connection enters the Idle state. After this response, the superior has no responsibilities to the subordinate with respect to the transaction.

e) [READONLY] - the subordinate no longer cares whether the transaction commits or aborts. The connection enters the Idle state. After this response, the superior has no responsibilities to the subordinate with respect to the transaction.

f) [ERROR] - the command was issued in the wrong state, or was malformed. The connection enters the Error state.

7) PULL. This command is only valid in Idle state. This command seeks to establish a superior/subordinate relationship in a transaction, with the primary party of the connection as the subordinate (i.e., he is pulling a transaction from the secondary party). Note that the entire value of <transaction string> (as defined in the Section "TIP Uniform Resource Locators" of [36]) must be specified as the transaction identifier.

   a) Input parameters:

      i) < connectionId >- connection identifier

      ii) < supTID > - superior's transaction identifier

      iii) < subTID > - subordinate's transaction identifier

   b) Possible responses:

      i) [PULLED] - the relationship has been established. Upon receipt of this response, the specified transaction becomes the current transaction of the connection, and the connection enters Enlisted state. Additionally, the roles of primary and secondary become reversed. (That is, the superior becomes the primary for the connection.)

      ii) [NOTPULLED] - the relationship has not been established (possibly, because the secondary party no longer has the requested transaction). The connection remains in Idle state.

      iii) [ERROR] - the command was issued in the wrong state, or was malformed. The connection enters the Error state.

8) PUSH. This command is valid only in the Idle state. It seeks to establish a superior/subordinate relationship in a transaction with the primary as the superior. Note that the entire value of <transaction string> (as defined in the Section "TIP Uniform Resource Locators") must be specified as the transaction identifier.

   a) Input parameters:

      i) <connectionId>- connection identifier

ii) <supTID > - superior's transaction identifier

b) Possible responses:

i) <subTID> - the relationship has been established, and the identifier by which the subordinate knows the transaction is returned. The transaction becomes the current transaction for the connection, and the connection enters Enlisted state.

ii) [ALREADYPUSHED] - the relationship has been established, and the identifier by which the subordinate knows the transaction is returned. However, the subordinate already knows about the transaction, and is expecting the two-phase commit protocol to arrive via a different connection. In this case, the connection remains in the Idle state.

c) [NOTPUSHED] - the relationship could not be established. The connection remains in the Idle state.

d) [ERROR] - the command was issued in the wrong state, or was malformed. The connection enters Error state.

9) QUERY. This command is valid only in the Idle state. A subordinate uses this command to determine whether a specific transaction still exists at the superior.

a) Input parameters:

i) <superior's transaction identifier>

b) Possible responses:

c) [QUERIEDEXISTS] - the transaction still exists. The connection remains in the Idle state.

d) [QUERIEDNOTFOUND] - the transaction no longer exists. The connection remains in the Idle state.

e) [ERROR] - the command was issued in the wrong state, or was malformed. The connection enters Error state.

10) RECONNECT. This command is valid only in the Idle state. A superior uses the command to re-establish a connection for a transaction, when the previous connection was lost during Prepared state.

a) Input parameters:

i) <connectionId>- connection identifier

ii) <subordinate's transaction identifier>

b) Possible responses:

i) [RECONNECTED] - the subordinate accepts the reconnection. The connection enters Prepared state.

ii) [NOTRECONNECTED] - the subordinate no longer knows about the transaction. The connection remains in Idle state.

iii) [ERROR] - the command was issued in the wrong state, or was malformed. The connection enters Error state.

Appendix 6 - Tmmap Schema

The XSD schema for defining mapping of TIP commands.

```
<xsd:schema
  targetNamespace="http://schemas.xmlsoap.org/wsdl/tmMap/"
   xmlns:tmmap="http://schemas.xmlsoap.org/wsdl/tmMap/"
  xmlns:wstl="http://schemas.xmlsoap.org/wsdl/wstl/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/"
              schemaLocation="http://localhost/Schemas/wsdl.xsd"/>
  <xsd:element name="definitions" type="tmmap:definitionsType">
    <xsd:key name="tmSrvMapKey">
      <xsd:selector XPath="tmmap:tmSrvMap"/>
      <xsd:field XPath="@name"/>
    </xsd:key>
    <xsd:key name="tmSrvPortTypeKey">
      <xsd:selector XPath="tmmap:tmSrvMap"/>
      <xsd:field XPath="@targetTmSrvPortType"/>
    </xsd:key>
    <xsd:keyref name="tmSrvPortTypeKeyRef" refer="wsdl:portType">
      <xsd:selector XPath="tmmap:tmSrvMap"/>
      <xsd:field XPath="@targetTmSrvPortType"/>
    </xsd:keyref>
  </xsd:element>
  <xsd:complexType name="definitionsType">
    <xsd:complexContent>
      <xsd:extension base="wsdl:documented">
        <xsd:sequence>
          <xsd:element ref="wsdl:import" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="tmmap:tmSrvMap" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="targetNamespace" type="xsd:anyURI" use="optional"/>
        <xsd:attribute name="name" type="xsd:NMTOKEN" use="optional"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="import" type="wsdl:importType"/>
  <xsd:element name="tmSrvMap" type="tmmap:tmSrvType">
    <xsd:annotation>
      <xsd:documentation>
        A mapping between the standard transaction manager
      </xsd:documentation>
      <xsd:documentation>
        service and the actual local transaction manager service
      </xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:complexType name="tmSrvType">
    <xsd:complexContent>
      <xsd:extension base="wsdl:documented">
        <xsd:sequence>
          <xsd:element ref="tmmap:abort"/>
          <xsd:element ref="tmmap:commit"/>
          <xsd:element ref="tmmap:prepare"/>
          <xsd:element ref="tmmap:open"/>
          <xsd:element ref="tmmap:close"/>
          <xsd:element ref="tmmap:push"/>
          <xsd:element ref="tmmap:pull"/>
          <xsd:element ref="tmmap:query"/>
          <xsd:element ref="tmmap:reconnect"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:NCName" use="required"/>
        <xsd:attribute name="targettmSrvPortType" type="xsd:QName" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="abort" type="tmmap:abortType">
    <xsd:annotation>
      <xsd:documentation>The abort command mapping</xsd:documentation>
    </xsd:annotation>
```

```
</xsd:element>
<xsd:complexType name="abortType">
  <xsd:complexContent>
    <xsd:extension base="tmmap:operationType">
      <xsd:sequence>
        <xsd:element name="inputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="connectionId" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="outputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="aborted" type="tmmap:paramLinkType"/>
                  <xsd:element name="error" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="operationType">
  <xsd:complexContent>
    <xsd:extension base="wsdl:documented">
      <xsd:attribute name="targetOperation" type="xsd:QName" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="msgLinkType">
  <xsd:annotation>
    <xsd:documentation>
      Defines the mapping between the standard transaction command
    </xsd:documentation>
    <xsd:documentation>
      and the actual local transaction command
    </xsd:documentation>
  </xsd:annotation>
  <xsd:attribute name="targetmsg" type="xsd:QName" use="required"/>
</xsd:complexType>
<xsd:complexType name="paramLinkType">
  <xsd:annotation>
    <xsd:documentation>
        Defines the mapping between the standard transaction command
    </xsd:documentation>
    <xsd:documentation>
      parameter and the actual local transaction command parameter
    </xsd:documentation>
  </xsd:annotation>
  <xsd:simpleContent>
    <xsd:extension base="xsd:string">
      <xsd:attribute name="targetParam" type="tmmap:XPathLinkType" use="required"/>
      <xsd:attribute name="responseValue" type="xsd:string" use="optional"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
<xsd:simpleType name="XPathLinkType">
  <xsd:annotation>
    <xsd:documentation>
        A subset of XPath expressions for use in XPathLinks for
        parameters mapping
    </xsd:documentation>
    <xsd:documentation>
      A utility type, not for public use
    </xsd:documentation>
  </xsd:annotation>
  <xsd:restriction base="xsd:string"/>
```

```
</xsd:simpleType>
<xsd:element name="commit" type="tmmap:commitType">
  <xsd:annotation>
    <xsd:documentation>The commit command mapping</xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:complexType name="commitType">
  <xsd:complexContent>
    <xsd:extension base="tmmap:operationType">
      <xsd:sequence>
        <xsd:element name="inputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="connectionId" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="outputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="commited" type="tmmap:paramLinkType"/>
                  <xsd:element name="aborted" type="tmmap:paramLinkType"/>
                  <xsd:element name="error" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="prepare" type="tmmap:prepareType">
  <xsd:annotation>
    <xsd:documentation>The prepare command mapping</xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:complexType name="prepareType">
  <xsd:complexContent>
    <xsd:extension base="tmmap:operationType">
      <xsd:sequence>
        <xsd:element name="inputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="connectionId" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="outputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="prepared" type="tmmap:paramLinkType"/>
                  <xsd:element name="aborted" type="tmmap:paramLinkType"/>
                  <xsd:element name="error" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="open" type="tmmap:openType">
  <xsd:annotation>
```

```
    <xsd:documentation>The open command mapping</xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:complexType name="openType">
  <xsd:complexContent>
    <xsd:extension base="tmmap:operationType">
      <xsd:sequence>
        <xsd:element name="inputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">

                <xsd:sequence>
                  <xsd:element name="lowestProtocolVersion" type="tmmap:paramLinkType"/>
                  <xsd:element name="highestProtocolVersion" type="tmmap:paramLinkType"/>
                  <xsd:element name="setTimeOut" type="tmmap:paramLinkType"/>
                  <xsd:element name="suptmurl" type="tmmap:paramLinkType"/>
                  <xsd:element name="alternateSuptmurl" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="outputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="connectionId" type="tmmap:paramLinkType"/>
                  <xsd:element name="error" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="close" type="tmmap:closeType">
  <xsd:annotation>
    <xsd:documentation>The close command mapping</xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:complexType name="closeType">
  <xsd:complexContent>
    <xsd:extension base="tmmap:operationType">
      <xsd:sequence>
        <xsd:element name="intputmsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="connectionId" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="outputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="closeOk" type="tmmap:paramLinkType"/>
                  <xsd:element name="error" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="push" type="tmmap:pushType">
  <xsd:annotation>
```

```
    <xsd:documentation>The push command mapping</xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:complexType name="pushType">
  <xsd:complexContent>
    <xsd:extension base="tmmap:operationType">
      <xsd:sequence>
        <xsd:element name="inputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="connectionId" type="tmmap:paramLinkType"/>
                  <xsd:element name="supTxId" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="outputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="subTxId" type="tmmap:paramLinkType"/>
                  <xsd:element name="alreadyPushed" type="tmmap:paramLinkType"/>
                  <xsd:element name="notPushed" type="tmmap:paramLinkType"/>
                  <xsd:element name="error" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="pull" type="tmmap:pullType">
  <xsd:annotation>
    <xsd:documentation>The pull command mapping</xsd:documentation>
  </xsd:annotation>
</xsd:element>
<xsd:complexType name="pullType">
  <xsd:complexContent>
    <xsd:extension base="tmmap:operationType">
      <xsd:sequence>
        <xsd:element name="outputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="connectionId" type="tmmap:paramLinkType"/>
                  <xsd:element name="subTxId" type="tmmap:paramLinkType"/>
                  <xsd:element name="supTxId" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="inputMsg">
          <xsd:complexType>
            <xsd:complexContent>
              <xsd:extension base="tmmap:msgLinkType">
                <xsd:sequence>
                  <xsd:element name="pulled" type="tmmap:paramLinkType"/>
                  <xsd:element name="notPulled" type="tmmap:paramLinkType"/>
                  <xsd:element name="error" type="tmmap:paramLinkType"/>
                </xsd:sequence>
              </xsd:extension>
            </xsd:complexContent>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="query" type="tmmap:queryType">
```

```
    <xsd:annotation>
      <xsd:documentation>The query command mapping</xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:complexType name="queryType">
    <xsd:complexContent>
      <xsd:extension base="tmmap:operationType">
        <xsd:sequence>
          <xsd:element name="outputMsg">
            <xsd:complexType>
              <xsd:complexContent>
                <xsd:extension base="tmmap:msgLinkType">
                  <xsd:sequence>
                    <xsd:element name="supTxId" type="tmmap:paramLinkType"/>
                  </xsd:sequence>
                </xsd:extension>
              </xsd:complexContent>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="inputMsg">
            <xsd:complexType>
              <xsd:complexContent>
                <xsd:extension base="tmmap:msgLinkType">
                  <xsd:sequence>
                    <xsd:element name="queryExists" type="tmmap:paramLinkType"/>
                    <xsd:element name="queryNotFound" type="tmmap:paramLinkType"/>
                    <xsd:element name="error" type="tmmap:paramLinkType"/>
                  </xsd:sequence>
                </xsd:extension>
              </xsd:complexContent>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:element name="reconnect" type="tmmap:reconnectType">
    <xsd:annotation>
      <xsd:documentation>The reconnect command mapping</xsd:documentation>
    </xsd:annotation>
  </xsd:element>
  <xsd:complexType name="reconnectType">
    <xsd:complexContent>
      <xsd:extension base="tmmap:operationType">
        <xsd:sequence>
          <xsd:element name="inputMsg">
            <xsd:complexType>
              <xsd:complexContent>
                <xsd:extension base="tmmap:msgLinkType">
                  <xsd:sequence>
                    <xsd:element name="subTxId" type="tmmap:paramLinkType"/>
                  </xsd:sequence>
                </xsd:extension>
              </xsd:complexContent>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="outputMsg">
            <xsd:complexType>
              <xsd:complexContent>
                <xsd:extension base="tmmap:msgLinkType">
                  <xsd:sequence>
                    <xsd:element name="reconnected" type="tmmap:paramLinkType"/>
                    <xsd:element name="notReconnected" type="tmmap:paramLinkType"/>
                    <xsd:element name="error" type="tmmap:paramLinkType"/>
                  </xsd:sequence>
                </xsd:extension>
              </xsd:complexContent>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

Appendix 7 - WSTL Schema for the Mediator and the Remote Services

XSD schema for defining mediator and remote services with WSTL.

```
<xsd:schema targetNamespace="http://schemas.xmlsoap.org/wsdl/wstl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tmmap="http://schemas.xmlsoap.org/wsdl/tmMap/"
  xmlns:wstl="http://schemas.xmlsoap.org/wsdl/wstl/"
  elementFormDefault="qualified" attributeFormDefault="unqualified">

  <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/tmMap/"
              schemaLocation="http://localhost/Schemas/tmmap.xsd"/>
  <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/wstl/"
              schemaLocation="http://localhost/Schemas/wstl.xsd"/>

  <xsd:element name="mediatorServiceDefinitions"
               type="wstl:mediatorServiceDefinitionsType"/>
  <xsd:complexType name="mediatorServiceDefinitionsType">
    <xsd:sequence>
      <xsd:element ref="wstl:mediatorService" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="mediatorService" type="wstl:mediatorServiceType"/>
  <xsd:complexType name="mediatorServiceType">
    <xsd:sequence>
      <xsd:element ref="wstl:operation" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:NCName" use="required"/>
  </xsd:complexType>

  <xsd:element name="operation" type="wstl:operationType"/>
  <xsd:complexType name="operationType">
    <xsd:sequence>
      <xsd:choice>
        <xsd:group ref="wstl:one-way"/>
        <xsd:group ref="wstl:request-response"/>
        <xsd:group ref="wstl:solicit-response"/>
        <xsd:group ref="wstl:notification"/>
      </xsd:choice>
      <xsd:element ref="wstl:contentDescription" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:group name="one-way">
    <xsd:sequence>
      <xsd:element ref="wstl:inputMsg"/>
    </xsd:sequence>
  </xsd:group>

  <xsd:group name="request-response">
    <xsd:sequence>
      <xsd:element ref="wstl:inputMsg"/>
      <xsd:element ref="wstl:outputMsg"/>
      <xsd:element ref="wstl:faultMsg" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:group>

  <xsd:group name="solicit-response">
    <xsd:sequence>
      <xsd:element ref="wstl:outputMsg"/>
      <xsd:element ref="wstl:inputMsg"/>
      <xsd:element ref="wstl:faultMsg" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:group>

  <xsd:group name="notification">
    <xsd:sequence>
      <xsd:element ref="wstl:outputMsg"/>
    </xsd:sequence>
  </xsd:group>
```

```
<xsd:element name="outputMsg" type="wstl:msgType"/>
<xsd:element name="inputMsg" type="wstl:msgType"/>
<xsd:element name="faultMsg" type="wstl:faultMsgType"/>

<xsd:complexType name="msgType">
  <xsd:sequence>
    <xsd:element ref="wstl:param" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="param" type="wstl:paramType"/>
<xsd:complexType name="paramType">
  <xsd:attribute name="name" type="xsd:NCName" use="required"/>
  <xsd:attribute name="type" type="xsd:QName" use="required"/>
  <xsd:attribute name="use" use="optional">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="required"/>
        <xsd:enumeration value="optional"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>

<xsd:complexType name="faultMsgType">
  <xsd:attribute name="errorCode" type="xsd:string" use="required"/>
  <xsd:attribute name="description" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:element name="remoteService" type="wstl:remoteServiceType"/>
<xsd:complexType name="remoteServiceType">
  <xsd:sequence>
    <xsd:element ref="wstl:operationMap" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="id" type="xsd:string" use="required"/>
  <xsd:attribute name="medServ" type="xsd:QName" use="required"/>
  <xsd:attribute name="portType" type="xsd:QName" use="required"/>
</xsd:complexType>

<xsd:element name="operationMap" type="wstl:operationMapType"/>
<xsd:complexType name="operationMapType">
  <xsd:sequence>
    <xsd:element ref="wstl:inputMap" minOccurs="0"/>
    <xsd:element ref="wstl:outputMap" minOccurs="0"/>
    <xsd:element ref="wstl:faultMap" minOccurs="0"/>
    <xsd:element ref="wstl:contentDescription" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:NCName" use="required"/>
  <xsd:attribute name="ptOperation" type="xsd:QName" use="required"/>
  <xsd:attribute name="medOperation" type="xsd:QName" use="required"/>
</xsd:complexType>

<xsd:element name="inputMap" type="wstl:msgMapType"/>
<xsd:element name="outputMap" type="wstl:msgMapType"/>
<xsd:element name="faultMap" type="wstl:msgMapType"/>

<xsd:complexType name="msgMapType">
  <xsd:sequence>
    <xsd:element ref="wstl:paramMap" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:NCName" use="optional"/>
</xsd:complexType>

<xsd:element name="paramMap" type="wstl:paramMapType"/>
<xsd:complexType name="paramMapType">
  <xsd:sequence>
    <xsd:element ref="wstl:sourceParam" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="targetParam" type="tmmap:XPathLinkType" use="required"/>
  <xsd:attribute name="mapFunction" type="xsd:NCName" use="optional"/>
</xsd:complexType>

<xsd:element name="sourceParam">
  <xsd:complexType>
    <xsd:attribute name="XPath" type="tmmap:XPathLinkType" use="optional"/>
    <xsd:attribute name="fixed" type="xsd:string" use="optional"/>
    <xsd:attribute name="responseValue" type="xsd:string" use="optional"/>
```

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

  <xsd:element name="contentDescription">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="wstl:domain" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="medParam" type="tmmap:XPathLinkType" use="required"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="domain">
    <xsd:complexType>
      <xsd:attribute name="value" type="xsd:string" use="required"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Appendix 8 - The Car Mediator Service Example

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:lxsd="http://example.com/MediatorService/carReservation/Schema/"
  xmlns:rs="http://example.com/RemoteService/carReservation/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tmmap="http://schemas.xmlsoap.org/wsdl/tmMap/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.xmlsoap.org/wsdl/
  http://localhost/Schemas/mediator.xsd"
  xmlns:wstl="http://schemas.xmlsoap.org/wsdl/wstl/">
  <import namespace="http://schemas.xmlsoap.org/wsdl/wstl/"
  location="http://localhost/Schemas/mediator.xsd"/>
  <import namespace="http://example.com/RemoteService/carReservation/"
  location="http://localhost/Schemas/mediator.xsd"/>
  <types>
    <schema
      targetNamespace="http://example.com/MediatorService/carReservation/Schema/"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType name="locationType">
        <attribute name="country" type="string"/>
        <attribute name="state" type="string"/>
        <attribute name="city" type="string"/>
        <attribute name="office" type="string"/>
      </complexType>
      <complexType name="companiesType">
        <sequence>
          <element name="companyName" type="string" maxOccurs="unbounded"/>
        </sequence>
      </complexType>
    </schema>
  </types>
  <wstl:mediatorService id="msCarReservation">
    <wstl:operation name="carReserv">
      <wstl:inputMsg>
        <wstl:param name="preferredCompany" type="lxsd:companiesType" use="optional"/>
        <wstl:param name="pickupDate" type="xsd:date" use="required"/>
        <wstl:param name="pickupLocation" type="lxsd:locationType" use="required"/>
        <wstl:param name="pickupTime" type="xsd:time" use="required"/>
        <wstl:param name="dropOffDate" type="xsd:date" use="required"/>
        <wstl:param name="dropOffLocation" type="lxsd:locationType" use="required"/>
        <wstl:param name="dropOffTime" type="xsd:time" use="required"/>
      </wstl:inputMsg>
      <wstl:outputMsg>
        <wstl:param name="reservationCode" type="xsd:string"/>
        <wstl:param name="company" type="xsd:string"/>
      </wstl:outputMsg>
      <wstl:faultMsg errorCode="ERROR_100" description="No available cars"/>
      <wstl:faultMsg errorCode="ERROR_101"
        description="No available remote services matching this invocation"/>
      <wstl:faultMsg errorCode="ERROR_102" description="Communication failure"/>
      <wstl:contentDescription medParam="inputMsg/preferredCompany/@companyName">
        <wstl:domain value="Avis"/>
        <wstl:domain value="Hertz"/>
        <wstl:domain value="Localiza"/>
        <wstl:domain value="BrazilCar"/>
      </wstl:contentDescription>
      <wstl:contentDescription medParam="inputMsg/pickUpLocation/@country">
        <wstl:domain value="USA"/>
        <wstl:domain value="Brazil"/>
        <wstl:domain value="Netherland"/>
        <wstl:domain value="Portugal"/>
      </wstl:contentDescription>
    </wstl:operation>
    <wstl:operation name="cancelCarReserv">
      <wstl:inputMsg>
        <wstl:param name="company" type="xsd:string"/>
        <wstl:param name="reservationCode" type="xsd:string"/>
      </wstl:inputMsg>
      <wstl:outputMsg>
        <wstl:param name="reservationCode" type="xsd:string"/>
      </wstl:outputMsg>
      <wstl:faultMsg errorCode="103" description="invalid reservation code"/>
```

```
    <wstl:faultMsg errorCode="104" description="Communication failure"/>
  </wstl:operation>
</wstl:mediatorService>
<wstl:remoteService id="rsBrazilCar" medServ="tns:msCarReservation"
                    portType="rs:reservationSoap">
  <wstl:operationMap name="reservMap" ptOperation="rs:reservation"
                     medOperation="tns:carReserv">
    <wstl:inputMap>
      <wstl:paramMap targetParam="reservationSoapIn/reservation/pInput/pickupInfo/@dt">
        <wstl:sourceParam XPath="@pickupDate"/>
      </wstl:paramMap>
      <wstl:paramMap
          targetParam="reservationSoapIn/reservation/pInput/pickupInfo/@location"
          mapFunction="concatValues">
        <wstl:sourceParam XPath="pickupLocation/@state"/>
        <wstl:sourceParam XPath="pickupLocation/@city"/>
        <wstl:sourceParam XPath="pickupLocation/@office"/>
      </wstl:paramMap>
      <wstl:paramMap
          targetParam="reservationSoapIn/reservation/pInput/pickupInfo/@time">
        <wstl:sourceParam XPath="@pickupTime"/>
      </wstl:paramMap>
      <wstl:paramMap
          targetParam="reservationSoapIn/reservation/pInput/dropoffInfo/@dt">
        <wstl:sourceParam XPath="@dropoffDate"/>
      </wstl:paramMap>
      <wstl:paramMap targetParam="reservationSoapIn/reservation/pInput/
                                  dropoffInfo/@location"
          mapFunction="concatValues">
        <wstl:sourceParam XPath="dropoffLocation/@state"/>
        <wstl:sourceParam XPath="dropoffLocation/@city"/>
        <wstl:sourceParam XPath="dropoffLocation/@office"/>
      </wstl:paramMap>
      <wstl:paramMap
          targetParam="reservationSoapIn/reservation/pInput/dropoffInfo/@time">
        <wstl:sourceParam XPath="@dropoffTime"/>
      </wstl:paramMap>
    </wstl:inputMap>
    <wstl:outputMap>
      <wstl:paramMap targetParam="@reservationCode">
        <wstl:sourceParam
            XPath="reservationSoapOut/reservationResponse/@reservationResult"/>
      </wstl:paramMap>
      <wstl:paramMap targetParam="@company">
        <wstl:sourceParam fixed="BrazilCar"/>
      </wstl:paramMap>
    </wstl:outputMap>
    <wstl:faultMap>
      <wstl:paramMap targetParam="@ERROR_100">
        <wstl:sourceParam
            XPath="reservationSoapOut/reservationResponse/@reservationResult"
            responseValue="NO_AV_CARS"/>
      </wstl:paramMap>
    </wstl:faultMap>
    <wstl:contentDescription medParam="preferredCompany/@company">
      <wstl:domain value="carBrasil"/>
    </wstl:contentDescription>
    <wstl:contentDescription medParam="pickupLocation/@country">
      <wstl:domain value="Brasil"/>
    </wstl:contentDescription>
  </wstl:operationMap>
  <wstl:operationMap name="cancelReservMap" ptOperation="rs:cancelReservation"
              medOperation="tns:cancelCarReserv">
    <wstl:inputMap>
      <wstl:paramMap
          targetParam="cancelReservationSoapIn/cancelReservation/@reservationCode">
        <wstl:sourceParam XPath="@reservationCode"/>
      </wstl:paramMap>
    </wstl:inputMap>
    <wstl:outputMap>
      <wstl:paramMap targetParam="@cancelationCode">
        <wstl:sourceParam XPath="cancelReservationSoapOut/cancelReservationResponse/
                                  @cancelReservationResult"/>
      </wstl:paramMap>
      <wstl:paramMap targetParam="@company">
        <wstl:sourceParam fixed="BrazilCar"/>
      </wstl:paramMap>
```

```
      </wstl:outputMap>
      <wstl:faultMap>
        <wstl:paramMap targetParam="@ERROR_103">
          <wstl:sourceParam XPath="cancelReservationSoapOut/cancelReservationResponse/
                                   @cancelReservationResult"
              responseValue="INV_RES_CODE"/>
        </wstl:paramMap>
      </wstl:faultMap>
    </wstl:operationMap>
  </wstl:remoteService>
</definitions>
```

Appendix 9 - WSTL Schema for Specifying Compositions

XSD schema for defining the WSTL elements that specify Web services compositions.

```
<xsd:schema
  targetNamespace="http://schemas.xmlsoap.org/wsdl/wstl/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:wstl="http://schemas.xmlsoap.org/wsdl/wstl/"
  xmlns:tmmap="http://schemas.xmlsoap.org/wsdl/tmMap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/tmMap/"
             schemaLocation="http://localhost/Schemas/tmmap.xsd"/>
  <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/wstl/"
             schemaLocation="http://localhost/Schemas/mediator.xsd"/>
  <!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
  <xsd:element name="compositionDefinition" type="wstl:compositonType"/>
  <xsd:complexType name="compositonType">
    <xsd:choice maxOccurs="unbounded">
      <xsd:element ref="wstl:dataLink"/>
      <xsd:element ref="wstl:faultDataLink"/>
      <xsd:group ref="wstl:task"/>
    </xsd:choice>
    <xsd:attribute name="name" type="xsd:ID" use="optional"/>
  </xsd:complexType>
  <!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
  <xsd:element name="rule">
    <xsd:complexType>
      <xsd:attribute name="condition" type="wstl:XPathType" use="required"/>
    </xsd:complexType>
  </xsd:element>
  <!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
  <xsd:element name="dataLink" type="wstl:dataLinkType"/>
  <xsd:complexType name="dataLinkType">
    <xsd:complexContent>
      <xsd:extension base="wsdl:documented">
        <xsd:sequence>
          <xsd:group ref="wstl:grplink" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="wstl:rule" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:ID" use="optional"/>
        <xsd:attribute name="source" type="xsd:QName" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
  <xsd:group name="grplink">
    <xsd:choice>
      <xsd:element ref="wstl:link"/>
      <xsd:element ref="wstl:foreachLink"/>
      <xsd:element ref="wstl:variableLink"/>
    </xsd:choice>
  </xsd:group>
  <!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
  <xsd:element name="dataLinkRef" type="wstl:baseRefType"/>
  <xsd:element name="faultLinkRef" type="wstl:baseRefType"/>
  <xsd:complexType name="baseRefType">
    <xsd:attribute name="name" type="xsd:QName" use="required"/>
  </xsd:complexType>
  <!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
  <xsd:element name="link" type="wstl:linkType"/>
  <xsd:complexType name="linkType">
    <xsd:complexContent>
      <xsd:extension base="wsdl:documented">
        <xsd:sequence minOccurs="0" maxOccurs="unbounded">
          <xsd:group ref="wstl:grplink"/>
        </xsd:sequence>
        <xsd:attribute name="append" type="xsd:boolean" use="optional" default="true"/>
        <xsd:attribute name="target" type="wstl:XPathType" use="optional"/>
        <xsd:attribute name="selection" type="wstl:XPathType" use="optional"/>
      </xsd:extension>
```

```
  </xsd:complexContent>
</xsd:complexType>
<!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<xsd:element name="variableLink" type="wstl:variableLinkType"/>
<xsd:complexType name="variableLinkType">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
    <xsd:group ref="wstl:grplink"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:NCName" use="required"/>
  <xsd:attribute name="source" type="xsd:QName" use="optional"/>
</xsd:complexType>
<!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<xsd:element name="foreachLink" type="wstl:foreachLinkType"/>
<xsd:complexType name="foreachLinkType">
  <xsd:sequence maxOccurs="unbounded">
    <xsd:group ref="wstl:grplink"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:NCName" use="required"/>
  <xsd:attribute name="select" type="wstl:XPathType" use="required"/>
</xsd:complexType>
<!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<xsd:group name="task">
  <xsd:choice>
    <xsd:element ref="wstl:atomicTask"/>
    <xsd:element ref="wstl:compositeTask"/>
    <xsd:element ref="wstl:syncTask"/>
    <xsd:element ref="wstl:foreachTask"/>
  </xsd:choice>
</xsd:group>
<!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<xsd:element name="atomicTask" type="wstl:atomicTaskType"/>
<xsd:complexType name="atomicTaskType">
  <xsd:complexContent>
    <xsd:extension base="wstl:baseTaskType">
      <xsd:sequence>
        <xsd:element ref="wstl:dataLinkRef" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="wstl:dataLink" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="medService" type="xsd:QName" use="required"/>
      <xsd:attribute name="operation" type="xsd:NCName" use="required"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<xsd:complexType name="baseTaskType">
  <xsd:complexContent>
    <xsd:extension base="wsdl:documented">
      <xsd:sequence>
        <xsd:element ref="wstl:dependency" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:ID" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<xsd:element name="dependency" type="wstl:dependencyType"/>
<xsd:complexType name="dependencyType">
  <xsd:complexContent>
    <xsd:extension base="wsdl:documented">
      <xsd:sequence>
        <xsd:element ref="wstl:rule" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="type" type="wstl:executionStateType" use="required"/>
      <xsd:attribute name="source" type="xsd:QName" use="optional"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<!--+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
<xsd:simpleType name="executionStateType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="start-start"/>
    <xsd:enumeration value="commit-start"/>
    <xsd:enumeration value="weak-commit-start"/>
    <xsd:enumeration value="abort-start"/>
    <xsd:enumeration value="commit-compensate"/>
    <xsd:enumeration value="commit/abort-start"/>
  </xsd:restriction>
```

```
  </xsd:simpleType>
  <!--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
  <xsd:element name="compositeTask" type="wstl:compositeTaskType"/>
  <xsd:complexType name="compositeTaskType">
    <xsd:complexContent>
      <xsd:extension base="wstl:baseTaskType">
        <xsd:sequence>
          <xsd:element ref="wstl:messages"/>
          <xsd:element ref="wstl:taskRef" maxOccurs="unbounded"/>
          <xsd:element ref="wstl:mandatoryTask"/>
          <xsd:element ref="wstl:dataLink" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="wstl:dataLinkRef" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="wstl:faultDataLink" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="wstl:faultLinkRef" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
  <xsd:element name="foreachTask" type="wstl:foreachTaskType"/>
  <xsd:complexType name="foreachTaskType">
    <xsd:complexContent>
      <xsd:extension base="wstl:compositeTaskType">
        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="selection" type="wstl:XPathType" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
  <xsd:element name="messages">
    <xsd:complexType>
      <xsd:group ref="wstl:request-response"/>
    </xsd:complexType>
  </xsd:element>
  <!--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
  <xsd:element name="mandatoryTask" type="wstl:mandatoryTaskType"/>
  <xsd:complexType name="mandatoryTaskType">
    <xsd:complexContent>
      <xsd:extension base="wsdl:documented">
        <xsd:sequence>
          <xsd:element ref="wstl:taskRef" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <!--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->

  <xsd:element name="taskRef" type="wstl:baseRefType"/>
  <!--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
  <xsd:element name="faultDataLink" type="wstl:dataLinkType"/>
  <!--++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-->
  <xsd:element name="syncTask" type="wstl:syncTaskType"/>
  <xsd:complexType name="syncTaskType">
    <xsd:complexContent>
      <xsd:extension base="wstl:baseTaskType">
        <xsd:sequence>
          <xsd:element ref="wstl:messages"/>
          <xsd:element ref="wstl:dataLinkRef" minOccurs="0" maxOccurs="unbounded"/>
          <xsd:element ref="wstl:dataLink" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
  <xsd:simpleType name="XPathType">
    <xsd:restriction base="xsd:string"/>
  </xsd:simpleType>
</xsd:schema>
```

Appendix 10 - A Composition Specification Example

This WSTL document specifies a business transaction of planning a trip. The trip plan consists of a hotel reservation and a car reservation for researchers attending a conference. The hotel and car reservations must be scheduled according to the following constraints. First, the system must try to make a hotel reservation in the conference hotel. If this reservation fails, the system must try a hotel reservation in any other hotel belonging to the list of indicated hotels of the conference organization. If this reservation succeeds, the system must try a car reservation, but only if the total price of the hotel room was less than a fixed value. The trip plan will succeed if any hotel reservation succeeds despite of the car reservation result. The WSTL definitions of the mediator and remote services used by this composition can be found in Appendix 11.

```
<definitions
  xmlns:tsktp="http://example.com/Applications/TripPlan/"
  xmlns:lxsd="http://example.com/Applications/TripPlan/Schema/"
  xmlns:mstrip="http://example.com/MediatorService/msTrip/"
  xmlns:mstripxsd="http://example.com/MediatorService/msTrip/Schema/"
  xmlns:wstl="http://schemas.xmlsoap.org/wsdl/wstl/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tmmap="http://schemas.xmlsoap.org/wsdl/tmMap/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.xmlsoap.org/wsdl/
  http://localhost/Schemas/composition.xsd">
  <import  namespace="http://example.com/Mediator/msTrip/"
           location="http://localhost/Mediator/msTrip.wstl"/>

  <types>
    <schema
      targetNamespace="http://example.com/Applications/TripPlan/Schema/"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType   name="tripPlanType">
        <sequence>
          <element name="Person" type="mstripxsd:PersonType"/>
          <element name="originInfo" type="mstripxsd:infoType"/>
          <element name="destinationInfo" type="mstripxsd:infoType"/>
          <element name="preferredAirlineCompanies"
                   type="mstripxsd:companiesType" use="optional"/>
          <element name="indicatedHotels" type="indicatedHotelsType"/>
        </sequence>
        <attribute   name="conferenceHotel" type="string" use="optional"/>
        <attribute   name="maxPriceToCarReserv" type="string" use="optional"/>
      </complexType>
      <complexType   name="tripPlanResponseType">
        <sequence>
          <choice>
            <element name="flightResponse" type="mstripxsd:tripInfoResultType"/>
            <element name="trainResponse" type="mstripxsd:tripInfoResultType"/>
            <element name="busResponse" type="mstripxsd:tripInfoResultType"/>
          </choice>
          <element name="carResponse"
                   type="mstripxsd:carReservationInfoType" minOccurs="0"/>
          <element name="hotelResponse" type="mstripxsd:hotelReservationInfoType"/>
        </sequence>
      </complexType>
      <complexType   name="indicatedHotelsType">
        <sequence>
          <element name="hotelName" type="string"/>
        </sequence>
      </complexType>
    </schema>
  </types>
```

```
  <wstl:compositionDefinition>

<!-- ++++++++++++++ Data Link Definitions for Task Car Reservation ++++++++++++ -->

    <wstl:dataLink name="dlCarInfo" source="tsktp:tripPlan">
      <wstl:link   target="inputMsg/param[name='person']"
                   selection="inputMsg/param[name='tripInfo']/@person">
      </wstl:link>
      <wstl:link   target="inputMsg/param[name='pickupDate']"
                   selection="inputMsg/param[name='tripInfo']/originInfo/schedule/@date">
      </wstl:link>
      <wstl:link   target="inputMsg/param[name='pickupTime']"
                   selection="inputMsg/param[name='tripInfo']/originInfo/schedule/@time">
      </wstl:link>
      <wstl:link   target="inputMsg/param[name='pickupLocation']"
                   selection="inputMsg/param[name='tripInfo']/originInfo/location">
      </wstl:link>
      <wstl:link   target="inputMsg/param[name='dropOffDate']"
                   selection="inputMsg/param[name='tripInfo']/originInfo/schedule/@date">
      </wstl:link>
      <wstl:link   target="inputMsg/param[name='dropOffTime']"
                   selection="inputMsg/param[name='tripInfo']/originInfo/schedule/@time">
      </wstl:link>
      <wstl:link   target="inputMsg/param[name='dropOffLocation']"
                   selection="inputMsg/param[name='tripInfo']/originInfo/location">
      </wstl:link>
      <wstl:variableLink name="maxPrice">
        <wstl:link selection="inputMsg/param[name='tripInfo']/@maxPriceToCarReserv"/>
      </wstl:variableLink>
    </wstl:dataLink>
    <wstl:dataLink name="dlHotelRoomPrice" source="tsktp:anyHotelReservation">
      <wstl:variableLink name="hotelRoomPrice">
        <wstl:link selection="outputMsg/param[name='info']/@price"/>
      </wstl:variableLink>
    </wstl:dataLink>

<!-- ++++++++++++++++++       Atomic Task Car Reservation +++++++++++++++++ -->
    <wstl:atomicTask   id="carReservation"
                       medService="msTrip:msCarReservation"
                       operation="carReserv">
      <wstl:dependency type="commit-start" source="tsktp:anyHotelReservation">
        <wstl:rule condition="$hotelRoomPrice &lt; $maxPrice"/>
      </wstl:dependency>
      <wstl:dataLinkRef name="dlCarInfo"/>
      <wstl:dataLinkRef name="dlHotelRoomPrice"/>
    </wstl:atomicTask>

<!-- +++ Data Link Definitions for Tasks Any and Conference Hotel Reservation +++ -->

    <wstl:dataLink name="dlHotelInfo" source="tsktp:tripPlan">
      <wstl:link   target="inputMsg/param[name='responsible']"
             selection="inputMsg/param[name='tripInfo']/Person"/>
      <wstl:link   target="inputMsg/param[name='checkinDate']"
                   selection=  "inputMsg/param[name='tripInfo']
                               /originInfo/schedule/@date"/>
      <wstl:link   target="inputMsg/param[name='checkoutDate']"
                   selection=  "inputMsg/param[name='tripInfo']
                               /destinationInfo/schedule/@date"/>
      <wstl:link   target="inputMsg/param[name='location']"
                   selection=  "inputMsg/param[name='tripInfo']
                               /destinationInfo/location"/>
    </wstl:dataLink>
    <wstl:dataLink name="dlConferenceHotel" source="tsktp:tripPlan">
      <wstl:link   target="inputMsg/param[name='preferredHotels']/companyName[1]"
                   selection="inputMsg/param[name='conferenceHotel']"/>
    </wstl:dataLink>
    <wstl:dataLink name="dlAnyHotel" source="tsktp:tripPlan">
      <wstl:link   target="inputMsg/param[name='preferredHotel']"
                   selection="inputMsg/param[name='indicatedHotels']" />
    </wstl:dataLink>

<!-- ++++++++++++++++++       Atomic Task Conference Hotel Reservation +++++++++++ -->
    <wstl:atomicTask   id="conferenceHotelReservation"
                       medService="msTrip:msHotelReservation"
                       operation="hotelReserv">
      <wstl:dependency type="start-start"/>
```

```
      <wstl:dataLinkRef name="dlHotelInfo"/>
      <wstl:dataLinkRef name="dlConferenceHotel"/>
    </wstl:atomicTask>

<!-- ++++++++++++++++++       Atomic Task Any Hotel Reservation +++++++++++++++++++ -->
    <wstl:atomicTask   id="anyHotelReservation"
                       medService="msTrip:msHotelReservation"
                       operation="hotelReserv">
      <wstl:dependency type="abort-start" source="tsktp:conferenceHotelReservation"/>
      <wstl:dataLinkRef name="dlHotelInfo"/>
      <wstl:dataLinkRef name="dlHotel"/>
    </wstl:atomicTask>

<!-- ++++++++++++++ Data Link Definitions for Composite Task TripPlan +++++++++ -->

      <wstl:dataLink   name="dlConferenceHotelReservation"
                       source="tsktp:conferenceHotelReservation">
        <wstl:link target="outputMsg/param[name='tripPlanResponse']/hotelResponse"
                   selection="outputMsg/param[name='reservationInfo']"/>
        <wstl:rule condition="committed(tsktp:conferenceHotelReservation)"/>
      </wstl:dataLink>
      <wstl:dataLink   name="dlAnyHotelReservation"
                 source="tsktp:anyHotelReservation">
        <wstl:link target="outputMsg/param[name='tripPlanResponse']/hotelResponse"
                   selection="outputMsg/param[name='reservationInfo']"/>
        <wstl:rule condition="committed(tsktp:anyHotelReservation)"/>
      </wstl:dataLink>
      <wstl:dataLink   name="dlCarReservation"
                       source="tsktp:carReservation">
        <wstl:link   target="outputMsg/param[name='tripPlanResponse']/carResponse"
                     selection="outputMsg/param[name='reservationInfo']"/>
        <wstl:rule condition="committed(tsktp:carReservation)"/>
      </wstl:dataLink>

<!-- ++++++++++ Fault links definitions for Composite Task Trip Plan ++++++++++++ -->
      <!-- Task Errors with the same error code are automatic bound -->
      <!-- It is not necessary to define a faultlink element. -->
      <!-- For example, the fault message with error code = "ERROR_102" -->
      <!-- is automatically routed between any atomic Task and the -->
      <!-- composite task Trip Plan -->
      <wstl:faultDataLink name="flAnyHotelReservation"
                          source="tsktp:anyHotelReservation">
        <wstl:link target="faultMsg/param[errorCode='ERROR_500']"
                   selection="faultMsg/param[errorCode='ERROR_105']"/>
        <wstl:rule condition="aborted(tsktp:anyHotelReservation)"/>
      </wstl:faultDataLink>
      <wstl:faultDataLink name="flConferenceHotelReservation"
                          source="tsktp:conferenceHotelReservation">
        <wstl:link   target="faultMsg/param[errorCode='ERROR_500']"
                     selection="faultMsg/param[errorCode='ERROR_105']"/>
        <wstl:rule condition="aborted(tsktp:ConferenceHotelReservation)"/>
      </wstl:faultDataLink>

<!-- ++++++++++++++++++       Composite Task Trip Plan ++++++++++++++++++++++++++++++ -->
    <wstl:compositeTask id="tripPlan">
      <wstl:dependency type="start-start"/>
      <wstl:messages>
        <wstl:inputMsg>
          <wstl:param name="tripInfo" type="lxsd:tripPlanType"/>
        </wstl:inputMsg>
        <wstl:outputMsg>
          <wstl:param name="tripPlanResponse" type="lxsd:tripPlanResponseType"/>
        </wstl:outputMsg>
        <wstl:faultMsg errorCode="ERROR_500" description="No available rooms"/>
        <wstl:faultMsg errorCode="ERROR_102" description="Communication failure"/>
      </wstl:messages>
      <wstl:taskRef name="tsktp:carReservation"/>
      <wstl:taskRef name="tsktp:conferenceHotelReservation"/>
      <wstl:taskRef name="tsktp:anyHotelReservation"/>
      <wstl:mandatoryTask>
        <wstl:taskRef name="tsktp:conferenceHotelReservation"/>
        <wstl:taskRef name="tsktp:anyHotelReservation"/>
      </wstl:mandatoryTask>
      <wstl:dataLinkRef name="dlConferenceHotelReservation"/>
      <wstl:dataLinkRef name="dlAnyHotelReservation"/>
      <wstl:dataLinkRef name="dlCarReservation"/>
      <wstl:faultLinkRef name="flConferenceHotelReservation"/>
```

```
      <wstl:faultLinkRef name="flAnyHotelReservation"/>
      <wstl:faultLinkRef name="flCarReservation"/>
    </wstl:compositeTask>

  </wstl:compositionDefinition>
</definitions>
```

Appendix 11 - Example of Mediator and Remote Services Definitions

This appendix shows the WSTL definitions of the mediator and remote services used by the composition of Appendix 10.

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:lxsd="http://example.com/MediatorService/Reservations/Schema/"
  xmlns:rs="http://example.com/RemoteService/carReservation/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wstl="http://schemas.xmlsoap.org/wsdl/wstl/"
  xmlns:tmmap="http://schemas.xmlsoap.org/wsdl/tmMap/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://schemas.xmlsoap.org/wsdl/
                      http://localhost/Schemas/mediator.xsd">

  <import  namespace="http://schemas.xmlsoap.org/wsdl/wstl/"
           location="http://localhost/Schemas/mediator.xsd"/>
  <types>
    <schema
      targetNamespace="http://example.com/MediatorService/Reservations/Schema/"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType name="tripInfoType">
        <sequence>
          <element name="Person" type="PersonType"/>
          <element name="originInfo" type="infoType"/>
          <element name="destinationInfo" type="infoType"/>
          <element name="preferredCompanies" type="companiesType" use="optional"/>
        </sequence>
      </complexType>
      <complexType name="tripInfoResultType">
        <sequence>
          <element name="departureInfo" type="infoType"/>
          <element name="arrivalInfo" type="infoType"/>
        </sequence>
        <attribute name="name" type="string"/>
        <attribute name="departureGate" type="string"/>
        <attribute name="company" type="string"/>
        <attribute name="reservationCode" type="string"/>
        <attribute name="expirationDate" type="date"/>
        <attribute name="price" type="float"/>
      </complexType>
      <complexType name="tripInfoResultType">
        <sequence>
          <element name="info" type="tripInfoResultType"/>
          <element name="legs" type="legsType"/>
        </sequence>
      </complexType>
      <complexType name="legsType">
        <sequence>
          <element name="leg" type="legType"/>
        </sequence>
      </complexType>
      <complexType name="legType">
        <attribute name="flight" type="string"/>
        <attribute name="departureNode" type="string"/>
        <attribute name="arrivalNode" type="string"/>
      </complexType>
      <complexType name="nodeType">
        <attribute name="airport" type="string"/>
        <attribute name="time" type="dateTime"/>
        <attribute name="gate" type="string"/>
      </complexType>
      <complexType name="locationType">
        <attribute name="country" type="string"/>
        <attribute name="state" type="string"/>
        <attribute name="city" type="string"/>
      </complexType>
      <complexType name="officelocationType">
        <sequence>
          <element name="location" type="locationType"/>
        </sequence>
```

```
  <attribute name="office" type="string" use="optional"/>
</complexType>
<complexType name="dateTimeType">
  <attribute name="date" type="date"/>
  <attribute name="time" type="time"/>
</complexType>
<complexType name="infoType">
  <sequence>
    <element name="schedule" type="dateTimeType"/>
    <element name="location" type="locationType"/>
  </sequence>
</complexType>
<complexType name="companiesType">
  <sequence>
    <element name="companyName" type="string" maxOccurs="unbounded"/>
  </sequence>
</complexType>
<complexType name="PersonType">
  <sequence>
    <element name="credityCard" type="creditCardType"/>
  </sequence>
  <attribute name="name" type="string"/>
  <attribute name="age" type="string"/>
</complexType>
<complexType name="guestsType">
  <sequence>
    <element name="name" type="string" maxOccurs="4"/>
  </sequence>
</complexType>
<complexType name="creditCardType">
  <attribute name="number" type="string"/>
  <attribute name="expirationDt" type="date"/>
  <attribute name="Company" type="string"/>
</complexType>
<complexType name="carClassType">
  <sequence>
    <element name="class" type="carClassEnum" maxOccurs="4"/>
  </sequence>
</complexType>
<simpleType name="carClassEnum">
  <restriction base="string">
    <enumeration value="Economy"/>
    <enumeration value="Luxury"/>
    <enumeration value="Sport"/>
    <enumeration value="Family"/>
  </restriction>
</simpleType>
<complexType name="categoriesType">
  <sequence>
    <element name="category" type="hotelCategoryEnum" maxOccurs="5"/>
  </sequence>
</complexType>
<simpleType name="hotelCategoryEnum">
  <restriction base="string">
    <enumeration value="fiveStars"/>
    <enumeration value="fourStars"/>
    <enumeration value="threeStars"/>
    <enumeration value="twoStars"/>
    <enumeration value="oneStar"/>
  </restriction>
</simpleType>
<complexType name="hotelReservationInfoType">
  <sequence>
    <element name="location" type="locationType"/>
    <element name="checkin" type="dateTimeType"/>
    <element name="checkout" type="dateTimeType"/>
  </sequence>
  <attribute name="name" type="string"/>
  <attribute name="hotelName" type="string"/>
  <attribute name="street" type="string"/>
  <attribute name="reservationCode" type="string"/>
  <attribute name="amountCharged" type="float"/>
  <attribute name="price" type="float"/>
</complexType>
<complexType name="carReservationInfoType">
  <sequence>
    <element name="pickupLocation" type="locationType"/>
```

```
      </sequence>
      <attribute name="name" type="string"/>
      <attribute name="company" type="string"/>
      <attribute name="reservationCode" type="string"/>
      <attribute name="carClass" type="carClassEnum"/>
      <attribute name="price" type="float"/>
    </complexType>
  </schema>
</types>
<wstl:mediatorServiceDefinitions>
  <!-- ++++++++++++++++++ Mediator Service Car Reservation ++++++++++++++++++ -->
  <wstl:mediatorService id="msCarReservation">
    <wstl:operation name="carReserv">
      <wstl:inputMsg>
        <wstl:param name="person" type="lxsd:personType"/>
        <wstl:param name="preferredCompany" type="lxsd:companiesType" use="optional"/>
        <wstl:param name="preferredcarClass" type="lxsd:carClass" use="optional"/>
        <wstl:param name="pickupDate" type="xsd:date" use="required"/>
        <wstl:param name="pickupLocation" type="lxsd:officelocationType" use="required"/>
        <wstl:param name="pickupTime" type="xsd:time" use="required"/>
        <wstl:param name="dropOffDate" type="xsd:date" use="required"/>
        <wstl:param   name="dropOffLocation"
                      type="lxsd:officelocationType" use="required"/>
        <wstl:param name="dropOffTime" type="xsd:time" use="required"/>
      </wstl:inputMsg>
      <wstl:outputMsg>
        <wstl:param name="reservationInfo" type="lxsd:carReservationInfoType"/>
      </wstl:outputMsg>
      <wstl:faultMsg errorCode="ERROR_100" description="No available cars"/>
      <wstl:faultMsg errorCode="ERROR_101"
                description="No available remote services matching this invocation"/>
      <wstl:faultMsg errorCode="ERROR_102" description="Communication failure"/>
      <wstl:contentDescription medParam="inputMsg/param[@name='preferredCompany']
                                                       /companyName">
        <wstl:domain value="Avis"/>
        <wstl:domain value="Hertz"/>
        <wstl:domain value="Localiza"/>
        <wstl:domain value="BrazilCar"/>
      </wstl:contentDescription>
      <wstl:contentDescription medParam="inputMsg/param[@name='pickUpLocation']
                                                       /location/@country">
        <wstl:domain value="USA"/>
        <wstl:domain value="Brazil"/>
        <wstl:domain value="Netherland"/>
        <wstl:domain value="Portugal"/>
      </wstl:contentDescription>
    </wstl:operation>
    <wstl:operation name="cancelCarReserv">
      <wstl:inputMsg>
        <wstl:param name="company" type="xsd:string"/>
        <wstl:param name="reservationCode" type="xsd:string"/>
      </wstl:inputMsg>
      <wstl:outputMsg>
        <wstl:param name="cancelationCode" type="xsd:string"/>
      </wstl:outputMsg>
      <wstl:faultMsg errorCode="103" description="invalid reservation code"/>
      <wstl:faultMsg errorCode="104" description="Communication failure"/>
    </wstl:operation>
  </wstl:mediatorService>
  <!-- ++++++++++++++++++ Mediator Service Hotel Reservation ++++++++++++++++++ -->
  <wstl:mediatorService id="msHotelReservation">
    <wstl:operation name="hotelReserv">
      <wstl:inputMsg>
        <wstl:param name="responsible" type="lxsd:personType" use="required"/>
        <wstl:param name="guests" type="lxsd:guetsType" use="optional"/>
        <wstl:param name="checkinDate" type="xsd:date" use="required"/>
        <wstl:param name="checkoutDate" type="xsd:date" use="required"/>
        <wstl:param name="location" type="lxsd:locationType" use="required"/>
        <wstl:param name="preferredHotels" type="lxsd:companiesType" use="optional"/>
        <wstl:param name="prefCategory" type="lxsd:categoriesType" use="optional"/>
      </wstl:inputMsg>
      <wstl:outputMsg>
        <wstl:param name="reservationInfo" type="lxsd:hotelReservationInfoType"/>
      </wstl:outputMsg>
      <wstl:faultMsg  errorCode="ERROR_105" description="No available rooms"/>
      <wstl:faultMsg  errorCode="ERROR_101"
                 description="No available remote services matching this invocation"/>
```

```
    <wstl:faultMsg errorCode="ERROR_102" description="Communication failure"/>

    <wstl:contentDescription medParam="inputMsg/param[@name='pickUpLocation']
                                                    /location/@country">
      <wstl:domain value="USA"/>
      <wstl:domain value="Brazil"/>
      <wstl:domain value="Portugal"/>
    </wstl:contentDescription>
  </wstl:operation>
  <wstl:operation name="cancelHotelReserv">
    <wstl:inputMsg>
      <wstl:param name="hotel" type="xsd:string"/>
      <wstl:param name="reservationCode" type="xsd:string"/>
    </wstl:inputMsg>
    <wstl:outputMsg>
      <wstl:param name="cancelationCode" type="xsd:string"/>
    </wstl:outputMsg>
    <wstl:faultMsg errorCode="106" description="invalid Hotel reservation code"/>
    <wstl:faultMsg errorCode="104" description="Communication failure"/>
  </wstl:operation>
</wstl:mediatorService>

<!-- ++++++++++++++++++ Mediator Service Flight Reservation ++++++++++++++++++ -->
<wstl:mediatorService id="msFlightReservation">
  <wstl:operation name="flightReserv">
    <wstl:inputMsg>
      <wstl:param name="tripInfo" type="lxsd:tripInfoType" use="required"/>
    </wstl:inputMsg>
    <wstl:outputMsg>
      <wstl:param name="reservationInfo" type="lxsd:flightResultType"/>
    </wstl:outputMsg>
    <wstl:faultMsg errorCode="ERROR_107" description="No available flyghts"/>
    <wstl:faultMsg  errorCode="ERROR_101"
              description="No available remote services matching this invocation"/>
    <wstl:faultMsg errorCode="ERROR_102" description="Communication failure"/>
    <wstl:contentDescription medParam="inputMsg/param[@name='tripInfo']
                                                    /preferredCompanies/companyName">
      <wstl:domain value="United"/>
      <wstl:domain value="Varig"/>
      <wstl:domain value="TAM"/>
      <wstl:domain value="Continental"/>
    </wstl:contentDescription>
  </wstl:operation>
  <wstl:operation name="cancelFlightReserv">
    <wstl:inputMsg>
      <wstl:param name="company" type="xsd:string"/>
      <wstl:param name="reservationCode" type="xsd:string"/>
    </wstl:inputMsg>
    <wstl:outputMsg>
      <wstl:param name="cancelationCode" type="xsd:string"/>
    </wstl:outputMsg>
    <wstl:faultMsg errorCode="108" description="invalid Flight reservation code"/>
    <wstl:faultMsg errorCode="104" description="Communication failure"/>
  </wstl:operation>
</wstl:mediatorService>

<!-- ++++++++++++++++++ Mediator Service Train Reservation ++++++++++++++++++ -->
<wstl:mediatorService id="msTrainReservation">
  <wstl:operation name="trainReserv">
    <wstl:inputMsg>
      <wstl:param name="tripInfo" type="lxsd:tripInfoType" use="required"/>
    </wstl:inputMsg>
    <wstl:outputMsg>
      <wstl:param name="info" type="lxsd:tripInfoResultType"/>
    </wstl:outputMsg>
    <wstl:faultMsg errorCode="ERROR_109" description="No available tickets"/>
    <wstl:faultMsg  errorCode="ERROR_101"
              description="No available remote services matching this invocation"/>
    <wstl:faultMsg errorCode="ERROR_102" description="Communication failure"/>
  </wstl:operation>
</wstl:mediatorService>
<!-- ++++++++++++++++++ Mediator Service Bus Reservation ++++++++++++++++++ -->
<wstl:mediatorService id="msBusReservation">
  <wstl:operation name="busReserv">
    <wstl:inputMsg>
      <wstl:param name="tripInfo" type="lxsd:tripInfoType" use="required"/>
    </wstl:inputMsg>
```

```
      <wstl:outputMsg>
        <wstl:param name="info" type="lxsd:tripInfoResultType"/>
      </wstl:outputMsg>
      <wstl:faultMsg  errorCode="ERROR_100"
                 description="No available tickets"/>
      <wstl:faultMsg  errorCode="ERROR_101"
                 description="No available remote services matching this invocation"/>
      <wstl:faultMsg  errorCode="ERROR_102"
                 description="Communication failure"/>
      <wstl:contentDescription medParam="inputMsg/param[@name='tripInfo']
                                          /destinationInfo/infoType/location/@country">
        <wstl:domain value="Brazil"/>
      </wstl:contentDescription>
      <wstl:contentDescription medParam="inputMsg/tripInfo/arrivalInfo/@country">
        <wstl:domain value="Brazil"/>
      </wstl:contentDescription>
    </wstl:operation>
  </wstl:mediatorService>
 </wstl:mediatorServiceDefinitions>
</definitions>
```