Handling Dissimilarities of Autonomous Equivalent Web Services

Valdino Azevedo, Marta Mattoso, Paulo F. Pires

[1] COPPE- System Engineering and Computer Science Program
DCC-IM/NCE[2]
Federal University of Rio de Janeiro
P.O. Box 68511, Rio de Janeiro, RJ, 21945-970, Brazil
Fax: +55 21 22906626
{valdino, marta, pires}@cos.ufrj.br

Abstract: The web services technology provides an essential building block for dynamic e-business, facilitating program-to-program interaction and the composition of new services. However, the need to specify service ports (or interfaces) in a service composition may constraint the usage of such technology. Given the existence of several semantically equivalent services on the Internet, we need a more flexible and loosely coupled way to include services in compositions. Instead of including specific services, a composition should deal with *service classes*, which group services with the same semantic functionality. During runtime, one or more services inside a service class can be scheduled to run, offering a dynamic mechanism for service execution. However, aggregating autonomous services that provide the same semantic functionality involves handling their dissimilarities such as transaction support, quality of service, message format and application domain. In this paper, we present mediation services for building classes of Web services. Such mediation layer is responsible for aggregating semantically equivalent services, thus providing a homogenized view of them, treating dissimilarities that may exist. Semantically equivalent services are grouped on service classes and compositions are specified on top of these classes. We also present an execution model that supports dynamic service executions by choosing services inside a service class based on their quality properties.

1 Introduction

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. In such an environment, it is likely to be found many different Web services, yet offering the same semantic functionality, i.e., services providing the same operations but having distinct WSDL interfaces. For example, consider two Web services providing hotel reservation from two different hotels. Both services may have different input message formats, however they provide the same kind of service, hotel reservation. These services can be considered equivalent Web services with respect to their functionalities.

It is important that a system, willing to offer value-added services, provides mechanisms for aggregating those equivalent Web services, which 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. Since a service within a Web service composition may be implemented by several available equivalent services, without mechanisms for homogenizing such equivalent, but heterogeneous Web services, it would be very unlikely that one could take some benefit when composing value-added services.

Web services can be heterogeneous in four different levels: (i) structural dissimilarities (i.e., message formats), (ii) content (i.e.: domains of their input and/or output objects), (iii) transactional support, and (iv) quality characteristics. To homogenize such semantically equivalent, though heterogeneous, Web services, all these dissimilarities must be handled. One way of dealing with Web services heterogeneity is to provide a layer of software components that provides mechanisms to homogenize heterogeneous Web services.

The problem of solving structural dissimilarities of services is similar to the problem of solving data dissimilarities occurring in mediator systems[22] and heterogeneous database systems[6,11,18]. Other works address the problem of solving structural and content dissimilarities in the Web through Wrapper-mediator technology [6]. However, such techniques cannot be directly used to solve structural dissimilarities of services since the problem of solving data dissimilarities is commonly related to the homogenization of different (data) schemas while the problem of solving structural dissimilarities of Web services is related the homogenization of different service interfaces (message formats).

Besides structural dissimilarities, Web services can differ regarding their 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 world wide reservations. Whether the content capability of Web services is available, it is possible select only those Web services capable of handling a specific execution, avoiding unnecessary calls to other Web services.

Since Web services are strictly autonomous units, they may expose different types of transaction support, even none support. Therefore, to provide a homogenized view of autonomous services, a service class must somehow expose a uniform transaction interface even though its aggregated services support dissimilar transaction mechanisms. In the multidatabase research area[13], there are transaction models that provide mechanisms to support the integration of data repositories with heterogeneous transaction support. However, these models have been designed with the assumption that all processing entities provide support for a set of transaction facilities such as supporting a two-phase commit interface[8], which restricts 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. Moreover, these models 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.

Equivalent Web services may be also different regarding their quality capabilities, like security mechanisms, response time and monetary cost. Different users will probably have distinct expectations about services behavior, i.e., one user can specify that only secure services can be executed, while other can specify that the service with the lowest response time should be invoked, independently of its monetary cost. Therefore, a layer of homogenized services should allow a client application to inform its expectations about how services should behave, indicating what quality characteristics a service must present. Only services supporting all quality aspects indicated should be scheduled for execution.

The main contribution of this work is to show how database mediation technologies can be adapted to solve Web Services heterogeneity for aggregating equivalent Web services. In previous works [10,12], we presented the specification of the WebTransact framework showing how reliable Web services compositions can be specified through such a framework. In this paper, we focus on the problem of homogenizing equivalent Web services, showing how this problem is solved within the WebTransact. Details on how transactional, message formats, content and quality dissimilarities are handled are shown. We also present the WebTransact Execution Model, which provides a runtime environment for scheduling a (sub)set of semantically equivalent Web services inside a service class to be executed. However, in this paper we concentrate on how such an execution model makes use of WebTransact mechanisms for homogenizing Web services. Implementation details as well as different strategies for dynamic web services executions supported by the WebTransact Execution Model are discussed in [1].

The remainder of this work is organized as follows. In Section 2, we present a general picture of the WebTransact architecture. Next, it is described how dissimilarities of Web services are solved, with examples being shown about how mediation among heterogeneous web services occurs. In Section 4, it is presented the WebTransact Execution Model. Finally, related work and concluding remarks are shown in Sections 5 and 6.

2 The WebTransact Architecture

As shown in Figure 1, WebTransact enables Web service composition by adopting a multilayered architecture of several specialized components [10,12]. 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. Besides resolving structural and content conflicts, remote services also provide information on the interface and the transaction behavior supported by Web services.

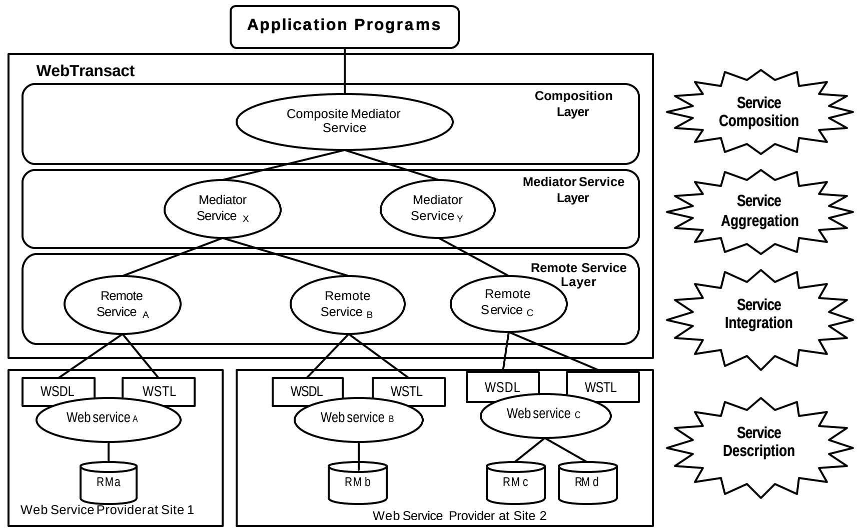


Fig.1. The WebTransact framework architecture

WebTransact integrates Web services through two XML-based languages: Web Service Description Language (WSDL) [20], which is the current standard for describing Web service interfaces, and Web Service Transaction Language (WSTL) [10], which is our proposal for describing the transaction support of heterogeneous Web services. WSTL is built on top of WSDL extending it with functionalities for enabling the transaction composition of Web services. Through WSDL, a remote service understands how to interact with a Web service. Through WSTL, a remote service knows the specific transaction support of a Web service. This language is also used to specify 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 service compositions.

3 Overcoming Web Services Heterogeneity

The WebTransact framework provides mechanisms for mediating Web services dissimilarities. Transactional dissimilarities are handled through the usage of a specific model for transaction management (specified through the WSTL language). Structural and content dissimilarities are resolved using the mediator technology, where mediators aggregate semantically equivalent web services and expose a single interface to be used in compositions. Finally, quality dissimilarities of Web services are treated by the WebTransact Execution Model, that selects one or more semantically equivalent Web services inside a service class to be scheduled to run, based on its quality characteristics and on constraints specified by a client application.

3.1 Resolving Transactional Dissimilarities of Web Services

Besides its signature, an operation of a specific Web service in WebTransact has a well-defined *transaction behavior*. The transaction behavior defines the level of transaction support that a given service exposes. In order to accommodate different levels of transaction support, four types of transaction behavior of services were defined (which are adaptations from distributed transactional control [15,23] for the Web services technology): *compensable*, *virtual-compensable*, *retriable*, and *pivot*. An operation is *compensable* if, after its execution, its effects can be undone by the execution of another operation. Therefore, for each compensable operation, it must be specified which other operation has to be executed in order to undo its effects. The *virtual-compensable* operation represents all remote operations whose underlying system supports the standard 2PC protocol [8]. Services offering such operations 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 operation. An operation is *retriable*, if it is guaranteed that it will succeed after a finite set of repeated executions. An operation is *pivot*, if it is neither retriable nor compensable.

Like remote services 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. The most restrictive transaction behavior is the pivot, followed by the retriable transaction behavior, while both the compensable and virtual-compensable transaction behaviors are the least restrictive transaction behavior. The concept of least/most restrictive transaction behavior defines whether a mediator service operation can participate in a given composition execution. A mediator service operation, aggregating at least one remote operation that is compensable (or virtual-compensable), can participate in any composition execution. On the other hand, a mediator service operation aggregating only pivot (or retriable) operations can participate only in compositions that call this mediator operation after it reaches a state where it is ready to successfully commit.

In this Section, we have only shown the main ideas behind the transaction behavior concept. Examples of WSTL documents describing WSDL operation supporting different types of transaction behavior can be found in [10].

3.2 Resolving Structural 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. Since mediators aggregate equivalent 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 services. A *remote service* links a mediator service to a WSDL port type element. It provides *mapping information* between mediator service operations and port type operations, and specifies the *content description* of the remote service. The mapping information prescribes how input and output parameters of a remote service operation are constructed from the input and output parameters of its related mediator service operation. The content description specifies whether a remote service is able to execute a particular service invocation, specifying a domain of values for a specific operation parameter. In next Section, it is shown an example about how these mapping efforts are done.

3.2.1 Remote Service Integration Example

In this Section, we illustrate the definitions of a mediator service and the integration of a remote service using WSTL[10]. Figure 2 shows a WSTL definition of a mediator service (`msBookShopServices`) providing book shops services. This mediator service supports two operations: `GetBookPrice` (that returns a book price given an ISBN number) and `BuyBook` (for buying books). In next paragraphs, we will detail the `BuyBook` operation.

The `BuyBook` operation has six input parameters: `ISBN` (ISBN number of the book), `CardNumber`, `CardType`, `FirstName` and `LastName` (related to the credit card specified by a client) and `PreferredShop` (that specifies in which shop this buy will be made). The return value of the `BuyBook` operation (`RequestID`) is an identification of this buy. The `BuyBook` operation may return one fault message identified by the value of the attribute `faultMsg/@errorCode`. The input parameters `PreferredShop` and `CardType` have their content description defined. For `PreferredShop` parameter, the content description specifies that this parameter can accept only elements within the domain of values {`BookShopWorld`, `WebBookSell`, `BookExpress`}. For `CardType` parameter, the content description specifies that this attribute accepts the following values: {`American Express`, `Visa`, `Mastercard`}.

Web services interfaces may be different from the mediator service interface. Therefore, a mechanism that maps the interface of a mediator to an interface of a specific Web Service is necessary. In WebTransact, *remote services* definitions are used to perform these mapping mechanisms. Figure 3 shows an example of a remote service description for the `BookShopWorld` service, one of the services aggregated by the `msBookShopServices` mediator.

The remote service is named `rsBookShopWorld`. The qualified value `rs:shopSoap` of the attribute `remoteService/@portType` links the WSTL remote service `rsBookShopWorld` to the port type element, `shopSoap`, of the WSDL Web Service. The remote service must provide mapping information for operations of the Web service that has different signatures from the ones specified in the mediator interface. In this example, only the `BuyBook` operation of that port type element is mapped to its related operation in the mediator service.

```
<wstl:mediatorService id="msBookShopServices">
  <wstl:operation name="GetBookPrice">
       <wstl:inputMsg>
                      <wstl:param name="ISBN" type="xsd:string"/>
       </wstl:inputMsg>
       <wstl:outputMsg>
                      <wstl:param name ="price" type="xsd:float"/>
       </wstl:outputMsg>
  </wstl:operation>
  <wstl:operation name="BuyBook">
       <wstl:inputMsg>
               <wstl:param name="ISBN" type="xsd:string"/>
               <wstl:param name="CardNumber" type="xsd:string"/>
               <wstl:param name="CardType" type="xsd:string"/>
               <wstl:param name="FirstName" type="xsd:string"/>
               <wstl:param name="LastName" type="xsd:string"/>
               <wstl:param name="PreferredShop" type="xsd:string"/>
       </wstl:inputMsg>
       <wstl: outputMsg>
               <wstl:param name="RequestId" type="xsd:string"/>
       </wstl:outputMsg>
       <wstl:faultMsg errorCode="ERROR_102"description="Invalid card
       number."/>

       <wstl:contentDescription medParam="inputMsg/@PreferredShop ">
               <wstl:domain value="AmericanExpress"/>
               <wstl:domain value="Visa"/>
               <wstl:domain value="MasterCard"/>
       </wstl:contentDescription>
       <wstl:contentDescription medParam="inputMsg/@CardType">
               <wstl:domain value="BookShopWorld"/>
               <wstl:domain value="WebBookSell"/>
               <wstl:domain value="Boo kExpress"/>
       </wstl:contentDescription>
  </wstl:operation>
</wstl:mediatorService>
```

Fig.2. WSTL Schema fragment of mediator service msBookShopServices

The `operationMap` element named `BookMap` defines a set of parameter-mappings between message of the port type operation, `BookShopBuy` and the parameters of the mediator service operation, `BuyBook`.

The `inputMap` element prescribes how fields in the input message part of the port type operation, `BookShopBuy`, are constructed from the input parameters of the mediator service operation, `BuyBook`. The first `paramMap` element defines a parameter-mapping that has as target node the XPath expression[21] `shopSoapIn/@ISBNnumber`. There is only one source node for this parameter-mapping, defined by the XPath expression `@ISBN`. According to this mapping, the field `shopSoapIn/@ISBNnumber` of the input message part of the port type operation `BookShopBuy`, is constructed from the input parameter `ISBN` of the mediator service operation.

```
<wstl:remoteService id="rsBookShopWorld"
                medServ="tns:msBookShopServices" portType="rs:shopSoap">
    <wstl:operationMap name="BookMap"
               ptOperation="rs:BookShopBuy" medOperation="tns:BuyBook">
        <wstl:inputMap>
            <wstl:paramMap targetParam="shopSoapIn/@ISBNnumber">
                <wstl:sourceParam XPath="@ISBN"/>
            </wstl:paramMap>
            <wstl:paramMap targetParam="shopSoapIn/CreditCard/@number">
                <wstl:sourceParam XPath="@CardNumber"/>
            </wstl:paramMap>
            <wstl:paramMap targetParam="shopSoapIn/CreditCard/@type">
                <wstl:sourceParam XPath="@CardType"/>
            </wstl:paramMap>
            <wstl:paramMap targetParam="shopSoapIn/CreditCard/@NameonCard"
                mapFunction="concatValues">
                <wstl:sourceParam XPath="@FirstName"/>
                <wstl:sourceParam XPath="@LastName"/>
            </wstl:paramMap>
        </wst:inputMap>

        <wstl:outputMap>
            <wstl:paramMap targetParam="@Code">
                <wstl:sourceParam XPath="@result"/>
            </wstl:paramMap>
        </wstl:outputMap>
        <wstl:contentDescription medParam="@PreferredShop">
            <wstl:domain value="BookShopWorld"/>
        </wstl:contentDescription>
        <wstl:contentDescription medParam="@CardType">
            <wstl:domain value="Visa"/>
            <wstl:domain value="MasterCard"/>
        </wstl:contentDescription>
    </wstl:operationMap>
</wstl:remoteService>
```

Fig.3. WSTL Schema fragment of remote service rsBookShopWorld

Next parameter-mappings deal with credit card information. In the port type operation, `BookShopBuy`, there is a structure named `CreditCard` that encapsulates all information about credit card data. The parameters `CardNumber` and `CardType` of the mediator service operation are mapped to the structure fields `CreditCard/@number` and `CreditCard/@Type`, as specified in the second and third `paramMap` elements. The fourth `paramMap` element also handles credit card data, but presents a `mapFunction` element named "`concatValues`". This function concatenates both `FirstName` and `LastName` parameter values of the mediator operation in one single field of the structure (`CreditCard/@NameonCard`) of the port type operation.

The `outputMap` element of the `BookMap` element prescribes how output parameters of the mediator service operation are constructed from the fields in the output message of the port type operation. In the example, a direct map from parameter result of the mediator to the parameter code of the port type is done.

Finally, the last definition enclosed by the `BookMap` element is the content description for the port type operation, `BookShopBuy`. In the example, there are two `contentDescription` elements. The first is defined for the input parameter `@PreferredShop` of the mediator service operation. This content description specifies that the selected remote service is able to attend requests when the value of the parameter `@PreferredShop` of the mediator service operation is within the domain of values {`BookShopWorld`} (that is, it can only serve requests to buy books on `BookShopWorld`). The other `contentDescription` element is defined for the input parameter `CardType` 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 `CardType`, of the mediator service operation, is within the domain of values {`Visa`, `MasterCard`}. This means that these are the only credit card types accepted by this specific Web service.

3.3 Resolving Quality Dissimilarities of Web Services

Semantically equivalent Web services can differ on its non-functional aspects, like its quality characteristics and cost parameters A client application, wishing to execute a service, can specify which quality characteristics a service must present. Therefore, our layer of homogenized services should support this kind of restriction made by client applications.

An operation call in WebTransact (a call to a specific mediator operation) is represented as an XML file that, besides the name of the operation being executed and its parameters, indicates quality constraints made by a client application. Figure 4 shows the layout of such XML file.

```
<operation     name="…"     maxresponsetime="..."          maxprice="..."
               executionmode="..."          priority=″...″
               mandatorycompensate="..."    security="..."  >

       <param name="..."></param>
       ...
       <param name="..."</param>
</operation>
```

Fig.4. XML document representing a mediator operation call.

The following quality aspects can be defined in an operation call:

? *maxresponsetime*: Maximum amount of time desired in service execution. Only services whose response times are equal or less than this maximum response time can be scheduled for execution.

? *maxprice*: Maximum monetary cost that a client accepts to pay for a service execution. Only services whose monetary cost is equal or less than the cost specified by the client application can be chosen for execution.

? *executiomode*: The execution mode is based on the relation between response time and monetary cost of execution of a service. An application can choose among one of the following execution modes: “Minimize Monetary Cost”, “Minimize

Response Time", "Best Cost/Performance Relation" or "Broadcast". The selected execution mode directly influences the process of services scheduling, as will be shown in next section.

? *priority:* Corresponds to the desired priority in the service execution, being transmitted in the header of SOAP messages sent to Web services providers.

? *mandatorycompesate*: A boolean attribute indicating if the remote service operation must be compensable.

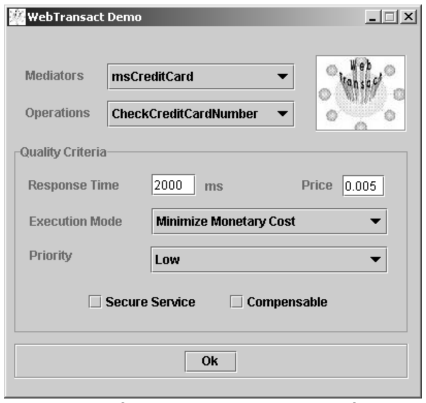
? *security*: A boolean attribute indicating if the remote service must present some security mechanism.

After showing how quality dissimilarities are handled, an execution model that supports service selection based on its quality aspects is needed. In next section will be presented the main steps of the WebTransact Execution Model, implemented in the WebTransact to support such service selection.

4 Scheduling Autonomous Equivalent Web Services

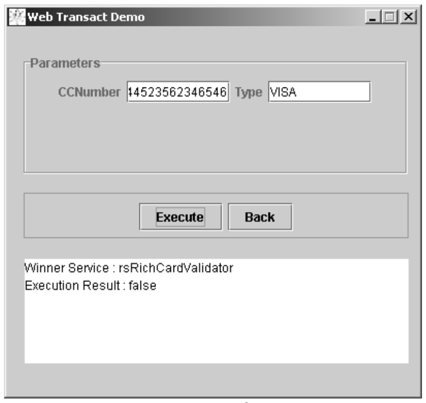
In its specification[10,12], the WebTransact framework did not have its main modules implemented and neither have an execution model that helps on the process of dynamic service selection. We have extended this framework with the creation of the WebTransact-EM (WebTransact Execution Model), that dynamically schedule services at runtime. In order to verify its results, we have implemented the main modules of the WebTransact framework together with the model itself.

Figure 5 shows the main screen of a prototype of the WebTransact-EM. A user can specify which constraints a service should support in order to be executed, as shown in Section 3.3. In figure 6, a user first enters the parameters needed for an operation execution. Then, the WebTransact-EM begins the process of service selection, scheduling and execution.



Mediators: msCreditCard
Operations: CheckCreditCardNumber
Response Time: 2000 ms
Price: 0.005
Execution Mode: Minimize Monetary Cost
Priority: Low

Fig. 5. Specifying quality constraints of an operation in WebTransact



CCNumber: 44523562346546
Type: VISA

```
Winner Service : rsRichCardValidator
Execution Result : false
```

Fig. 6. Execution results of an operation

First, an XML file as the one shown in Section 3.3 is constructed from the constraints and parameters specificied by the user. When the WebTransact framework receives such an operation call, it must decide which services inside a service class will be executed. Four different phases were identified to compose the WebTransact Execution Model: (i) selection of candidate services, (ii) ordering of these services based on an execution mode, (iii) scheduling of services and finally, (iv) the choice of a winner service. These phases will be shown in next paragraphs.

Candidate services represent the set of remote services aggregated by a given mediator service, which support some quality criteria specified by a client application. They are chosen based on the *content description* of services, presented in Section 3.2 and on the supported *quality aspects* of services, presented in Section 3.3. Only services that support quality criteria constraints indicated by a client application can be further scheduled to run.

Next, *candidate services* must be ordered based on some criteria derived from Web services quality characteristics. In WebTransact, they can be ordered by their response times, monetary costs or cost/performance relations, depending on the execution mode chosen by the composition specification.

Given a list of Web services already ordered, the scheduling process begins, taking into account the desired execution mode and the transactional behavior of each service. Depending on the execution mode, one of the following scheduling strategies will be used:

? *Minimize Response Time:* The choice of which services are scheduled to run is based on the response time only. Thus, it is independent of other costs such as the monetary cost involved (the service with the lowest response time is preferable). Response time can be minimized through parallel scheduling of all candidate services. However, only compensable services (the ones whose results can be undone) can be executed concurrently, since only one service will have its results committed and returned to a client application (being elected as the "winner service).

? *Minimize Monetary Cost:* In this case, the choice of which services must be scheduled to run will only consider monetary costs, the response time is not taken into account. Services are first ordered (in the previous step) according to its monetary costs and the candidate service with the lowest cost is the only one to be scheduled to run.

? *Best Cost/Performance Relation:* Now, both parameters are taken into account: the response time of a service and its monetary cost. If exists candidate services with no monetary cost (equal to zero), all services of this type are executed in parallel, in order to reach the best relation of cost and performance. Otherwise, after the ordering of candidate services based on their cost/performance relation, the service having this best relation is scheduled to run.

? *Broadcast:* This execution mode indicates that the client application has requested the execution of all candidate services and that it desires the results of all of them. Therefore, all candidate services are scheduled to run concurrently and their results are committed and returned to the client application.

After the scheduling phase, all services scheduled are executed. After services finish their executions, we need to choose a winner service (the one whose results will be committed, while the others will be aborted or compensated).

For a broadcast execution, all services are elected as winner services, and the results of all of them are returned. Otherwise, if only one service has already ended, this one will be the winner and all of the other services being executed (if they exist) will be aborted or compensated. If more than one service has ended the winner will be the one with the higher compensation cost (in order to avoid the high cost of this operation). After the selection of winner services, an execution in WebTransact is completed. The complete algorithms for service selection, scheduling and execution of the WebTransact -EM can be found in [1].

5 Related Work

Current frameworks that support service compositions [4,7,16], do not handle the inherent heterogeneity of Web services nor presents an execution model for dynamic services executions. We propose a solution for this problem through the specification and implementation of a mediator layer that homogenizes services interfaces and their transactional behaviors. Works in the bioinformatics area [14] also propose the mediator technology, in addition to the Web services Technology, to homogenize legacy applications. However, these works do not employ an execution model supporting dynamic service execution, and neither handle transactional, content and quality dissimilarities that may exist among these applications.

Existent works in the area of e-service composition (WSFL [9], XLANG [17], WSCL [19], BPEL4WS [5]) are concentrated in defining primitives for composing services and automating service coordination. However, these primitives for composition do not directly address the problems associated with the necessary *homogenization* of Web services. In this sense, mechanisms presented in this paper can complement such works.

More recently, other specifications were made for the definition of transaction languages for Web services, such as the WS-Transaction[3] together with the WS-Coordination[2] specifications. Several concepts that exist in the WSTL language [10] are also present in WS-Transaction and WS-Coordination, such as the usage of a Web service that acts as a coordinator for the execution of distributed transactions (mechanism used for virtual-compensable services in WSTL) and the concept of compensation activities to avoid resources locking for a great amount of time. However, the definition of a transaction behavior for each operation of a service, made through the WSTL language, makes simpler the transactional management of activities that do not use a protocol such as the 2PC [8], since they do not need to be related to a coordinator service. In this case, there must exist a platform (the WebTransact, for example) that handles the transactional management of such activities. Finally, as several concepts used in both specifications are similar, we believe that both languages (WS-Transaction and WSTL) can coexist without any compatibility problem if a service that relies on WSTL for transactional management wishes to communicate with another service that uses the WS-Transaction.

6 Conclusions

There is a real potential for appearing a huge number of semantically equivalent, but heterogeneous, Web services in the near future. Since such services will be published by autonomous companies, they may expose dissimilar interfaces and behavior. Therefore, the utilization of such heterogeneous Web services can be improved by a software layer responsible for mediating their dissimilarities. Such a mediation layer hides the dissimilarities and even the existence of many different equivalent Web services, providing a single common interface, which is used to build new business processes. Equivalent Web services can differ on several aspects, like their transactional support, their interfaces and their quality characteristics. Therefore, the mediation layer must somehow resolve these dissimilarities.

The main contribution of this work is to show how Web services dissimilarities are solved in *WebTransact*, a framework for reliable execution of Web services compositions. Transactional, structural, content and quality dissimilarities of Web services are handled through the creation of *service classes*, which is responsible for mediating service dissimilarities. A service class hides dissimilarities of semantically equivalent but different and autonomous Web services, exposing to client programs a homogenized view of such services. When a service class is invoked, the WebTransact Execution Model acts automatically choosing one or more of its semantically equivalent Web services for execution.

Through the mechanisms presented in this paper, a business process can be built using service classes instead of directly access specific Web service implementations. Since service classes aggregate a set of semantically equivalent Web services and handle all the inherent heterogeneity of them, developers of such business process are unburdened of dealing with the heterogeneity and the distribution of such services. Moreover, the use of service classes improves the robustness and reduces the response time and the impact of Web service changes in a business processes. Since a service class aggregate a set of semantically equivalent Web services, even when a subset of these services are unavailable, the service class can successfully execute if at least one of its aggregated services were available. Finally, since business process do not directly references Web service interfaces, changes in the latter does not directly affect the former (only the mapping information must change). It is important to note that the concept of service classes along with the WebTransact Execution Model can easily be used to aggregate value to other platforms, such as platforms for Web service composition [4,7,16]. Since service classes are regular Web services, they can be seamlessly incorporated in any platform adherent to the Web services technology.

References

1. Azevedo, V., Mattoso, M., Pires, P. F. , "Strategies for Dynamic Execution of Semantically Equivalent Web Services". In: Intenational Conference on Web Services – Europe (ICWS-Europe), Erfurt, Germany, September, 2003 (submitted).
2. Cabrera, F., Copeland, G., Freund, T., "Web Services Coordination (WS-Coordination)", 2002. Available at: http://www-106.ibm.com/developerworks/library/ws-coor

3. Cabrera, F., Copeland, G., Cox, B.,"Web Services Transaction (WS-Transaction)", 2002. Available at: http://www-106.ibm.com/developerworks/library/ws-transpec
4. Casati, F., Ilnicki, S., Jin, L., et al., "Adaptive and Dynamic Service Composition in eFlow". In: Proceedings of CaiSE 2000, Stockholm, Sweden, pp. 13-31, June 2000.
5. Curbera, F., Goland, Y., Klein, J., *et al.* "Business Process Execution Language for Web Services Version 1.0", 2002. Available at: http://www.ibm.com/developerworks/library/ws-bpel/
6. 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
7. Keidl, M., Seltzam, S., Stocker, K., Kemper, A., 2002 , "ServiceGlobe: Distributing E-Services Across the Internet". In: *Proc. of the 28th VLDB Conference*, Hong Kong, China, August, 2002.
8. 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
9. Leymann, F., "Web Services Flow Language (WSFL 1.0)". [http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf], May, 2001.
10. Pires, P., WebTransact: A Framework for Specifying and Coordinating Reliable Web Services Composition. Technical Report ES-578/02, COPPE/UFRJ, Brazil, 2002.
11. 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.
12. Pires, P.F., Mattoso, M., Benevides, M., "Building Reliable Web Services Compositions" In: Proc. of the NET.Object Days Conference (WS-RDS'02), Erfurt, Germany, 2002.
13. 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.
14. Rocco, D., Critchlow, T., *Discovery and Classification of BioInformatics Web Services*. Technical Report, Lawrence Livermore National Laboratory, September, 2002.
15. Schuldt, H., Alonso, G., Schek, H.J., "Concurrency Control and Recovery in Transactional Process Management". In: Proc. of the Symposium on Principles of Database Systems (PODS), Philadelphia, Pennsylvania, pp.316-326, 1999.
16. Sheng, Q., Benatallah, B., Dumas, M., Mak, E., "SELF-SERV: A Platform for Rapid Composition of Web Services in a Peer-to-Peer Environment". In: *Proc. of the 28th VLDB Conference*, Hong Kong, China, August, 2002.
17. Thatte, S., "XLANG: Web Services for Business Process Design". [http://www.gotdotnet. com/team/xml_wsspecs/xlang-c/default.htm], Microsoft Corporation, 2001.
18. 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.
19. W3C (World Wide Web Consortium) Note, "Web Services Conversation Language (WSCL) 1.0". [http://www.w3.org/TR/2002/NOTE-wscl10-20020314/], March 2001.
20. W3C (World Wide Web Consortium) Note, "Web Services Description Language (WSDL) 1.1". [http://www.w3.org/TR/2001/NOTE-wsdl-20010315], March 2001.
21. W3C (World Wide Web Consortium) Recommendation, "XML Path Language". [http://www.w3.org/TR/1999/REGXPath-19991116], November 1999.
22. Wiederhold, G., "Mediation in Information Systems", ACM Computing Surveys, v. 27, n. 2, pp. 265-267, 1995.
23. Zhang, A., Nodine, M.H., Bhargava, B.K., Bukhres, O., "Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems". In: Proc. of the ACM SIGMOD Conference, pp. 67-78, 1994.