Design Method for Interoperable Web Services

George Feuerlicht
Faculty of Information Technology,
University of Technology, Sydney,
P.O. Box 123 Broadway, Sydney, NSW 2007,
Australia
61 2 9514 1835
jiri@it.uts.edu.au

Sooksathit Meesathit
Faculty of Information Technology,
University of Technology, Sydney,
P.O. Box 123 Broadway, Sydney, NSW, 2007,
Australia
61 2 9514 4512
smeesath@it.uts.edu.au

ABSTRACT
The emergence of Web services provides an opportunity to address e-business application interoperability in the context of service-oriented computing. In this paper we discuss the benefits of the service-oriented approach to implementing e-business applications and identify the need for well-designed service interfaces to facilitate interoperability within application domains. We describe a service interface design method based on identifying elementary business function and converting standard message (document) formats into a set of corresponding service-interfaces. We then apply data engineering principles to refine the interface design, and show how data normalization applied to interface parameters can lead to minimization of coupling and maximization of cohesion of service operations. We illustrate our design approach using a travel application example based on the Open Travel Alliance (OTA) specification.

Categories and Subject Descriptors
D.2.2 [**Software engineering**]: Design tools and techniques – *Modules and interfaces*.

General Terms
Design

Keywords
Web services, service interface design, e-business interoperability

1. INTRODUCTION
Interoperability is a key requirement for inter-enterprise, e-business (electronic business) applications. The principal challenge is ensuring interoperability in an environment where participating organizations use disparate business semantics and technology platforms. Most existing approaches rely on standardization of business documents and inter-organizational business processes to allow automation of e-business. Early e-business approaches such as EDI (Electronic Data Interchange) [27] have not succeeded in addressing interoperability effectively, producing proprietary document formats typically shared among relatively small groups of companies. The proprietary nature of EDI necessitates complex translations to overcome differences in syntax, structure, and semantics of documents used by individual partner organizations, resulting in costly and inflexible point-to-point solutions. More recent approaches use XML formatted documents and Internet communication protocols (e.g. BizTalk [15]), but also suffer from limited scalability that characterizes point-to-point solutions [13]. Other e-business approaches avoid the problems associated with point-to-point solutions and rely instead on the interchange of business documents with pre-defined structure and semantics, and specification of standard business processes. For example, RosettaNet [21] and ebXML [4] attempt to reach agreement about common business semantics among a broad range of partner organizations (e.g. across entire industry domains) by defining common vocabularies of business objects (or components) and business protocols. Although successful in some industry domains (e.g. RosettaNet in the high technology industry domain) wide adoption of such standards has been limited by their complexity, inflexibility and high implementation costs.

In order to address the requirements of modern e-business applications in an environment characterized by a large number of autonomous partner organizations with independently evolving business semantics, the interoperability mechanisms must be both scalable and flexible. Scalability in this context is the ability to accommodate a large number of partner organizations with diverse business semantics without unduly increasing the complexity of the specification and at the same time avoiding the need for point-to-point transformations. Flexibility is needed to allow the evolution of the standard specification to accommodate changes in business processes and data semantics without impacting on existing applications.

In this paper we argue that the requirements of e-business applications cannot be satisfactorily addressed using an interoperability mechanism based on document interchange (section 2), and that the service-centric model provides a superior interoperability solution (section 3). Interoperability in the context of the service-centric approach is critically dependent on service interfaces used to expose business functionality. Poorly designed interfaces result in duplication of functionality, limited reuse and extensibility, and poor maintainability of applications. We describe a service interface design method based on principles of object-oriented programming and component design, and adapted for service interface design (section 4). We then apply data engineering principles to refine the interface design, and show how data normalization applied to interface parameters can lead to minimization of coupling and maximization of cohesion of service operations. We illustrate our design approach using a travel application example based on the Open Travel Alliance specification, showing how a document-centric specification can be transformed into s set of well-designed Web service interfaces. In conclusion (section 5) we summarize the main contributions of this paper and discuss the advantages and limitations of the proposed design approach.

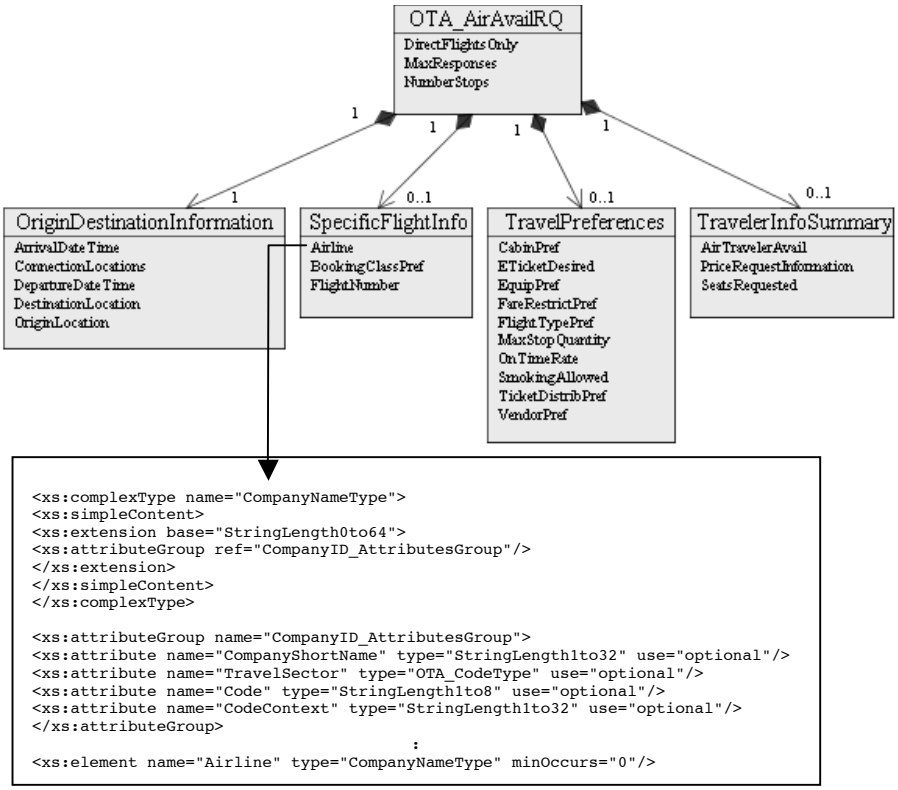

Figure 1. Simplified content of OTA_AirAvailRQ message.

2. DOCUMENT-CENTRIC INTEROPERABILITY MODEL

The main advantage of the document-centric approach is the ability to interoperate across heterogeneous environments, as documents can be transmitted as message payloads using a variety of messaging protocols (e.g. SMTP/POP, SOAP, or a proprietary EDI protocol) irrespective of the underlying technology platform. Standard document formats have been defined for various industry domains, for example the Open Travel Alliance (OTA) consortium specification [17] defines XML Schemas and corresponding usage scenarios for messages that support business activities in the travel industry. A large number of message formats addressing various aspects of travel business have been defined providing a comprehensive specification of information requirements for travel applications. Examples in this paper are loosely based on the OTA specification and use simplified subsets of OTA message structures in order to allow illustration of interface design concepts within the limited scope of this paper; full description of the OTA specification is available on: http://www.opentravel.org/. Comments made about OTA message structures are not intended as criticism of OTA design, but as general comments about the limitations of the document-centric approach in e-business applications.

2.1 Flight Booking Example

Consider the following flight booking scenario involving an airline and a travel agent. A travel agent requests flight availability information from an airline specifying the departure and destination city, the date of travel, and other relevant information. The airline responds with a collection of available flights; the agent then requests pricing information, and finally may proceed to book a selected flight. To implement this scenario OTA defines relevant message payloads using XML Schema. For example, the OTA Air Availability Request (OTA_AirAvailRQ) message illustrated in Figure 1 is a schema specification for a request for flight availability information. The OTA_AirAvailRQ message contains a large number of schema elements including origin and destination information, specific flight information, and passenger travel preferences. We only show simplified message content here; many elements are further structured as illustrated for the **Airline** element, resulting in a complex message structure designed to maximize the amount of information transmitted within a single message payload. The intention is to transmit all information required for processing the request in a single message so that it can be executed as one transaction without dependence on information transmitted in previous messages [17]. OTA specification is based on the request/response paradigm, and a corresponding Air Availability Response message (OTA_AirAvailRS) that contains information about flights matching the request criteria is also specified. Each message implicitly represents a complex business process that the receiving partly needs to decompose and map to local transactions.

2.2 Limitations of the Document-centric Approach

Using document interchange as the interoperability mechanism for e-business applications leads to message specifications with highly complex data structures. The complexity of message formats leads to corresponding complexity of applications, and consequently to high implementation costs. Implicit in the structure of the messages are various constraints, business rules, and control parameters. For example, the OTA_AirAvailRS message allows up to four flight segments for a given pair of origin (OriginLocation) and destination (DestinationLocation) locations. The control parameter DirectFlightsOnly (if set on), allows only a single flight segment between the origin and destination location. Furthermore, as noted above, individual messages are designed to be self-contained, i.e. to contain all the data required for a particular transaction (e.g. flight booking). This leads to duplication with the same data structures used in multiple messages.

Perhaps the most significant drawback of document-centric model is that externalizing data structures in the form of document schemas creates dependencies between partner applications. This makes the document-centric standard inflexible and difficult to evolve as changes in document specifications (i.e. modifications of document structure, such as inclusion of new elements, or removal of existing elements) directly impact on existing applications. As experience with EDI indicates, even in situations where there is initial agreement, message standards tend to diverge over time to reflect the specific requirements of individual partner organizations and industry sectors, resulting in incompatible document specifications that require extensive transformations to facilitate interoperability.

3. SERVICE-CENTRIC INTEROPERABILITY MODEL

Web services [31] provide e-business infrastructure where business partners interact by consuming partner services [22], [19], shifting the focus from connecting to a particular business partner to the definition of business interfaces that address specific business requirements and are available to all participating organizations [20]. In effect, Web services create a homogeneous runtime environment for e-business applications removing the need to use data interchange as the interoperability mechanism. Although Web services can transmit XML documents using the document style binding (so that, for example OTA messages could be implemented with SOAP as the transport mechanism) the main benefit of Web services is the ability to use programmatic techniques for the implementation of e-business applications, elevating the level of abstraction from document interchange to service invocation. Web services support for Remote Procedure Calls (RPCs) provides a mechanism for transparent execution of remote services irrespective of the underlying platform used by individual partners. Consequently, the problem of standardizing structure and semantics of documents can be reduced to a more manageable task of standardizing service interfaces. Important advantage of the service-centric approach is that service interfaces can be designed to significantly reduce coupling between applications resulting in improved scalability, and at the same time making applications more resilient to change.

We note that adopting the service-centric approach does not directly address the semantic issues of e-business interoperability. Agreement on semantics across a domain of interest (e.g. travel) is a pre-requisite for an interoperable e-business environment. Well-designed service interfaces address the equally important problem of externalizing complex and often redundant data structures that characterize the document-centric approach, and result in high levels of interdependency between applications. Adopting the service-centric approach transforms the problem of defining and maintaining complex document structures into a more manageable problem of standardizing domain-specific service interfaces. Moving away from an interoperability model based on document interchange and adopting a programmatic approach to e-business applications results in a higher level of abstraction associated with the use of application programming interfaces. As experience with programming APIs (Application Programming Interfaces) demonstrates, the benefits of standardized service interfaces include improved software reliability, reusability, extensibility, and maintainability and can lead to significant application development productivity gains. Importantly, using the service-centric approach evolution and refinement are supported via publishing new versions of existing interfaces, and by adding new interfaces, while maintaining the standard set of interfaces. This provides an effective mechanism for evolution that maintains existing (externalized) interface without any modifications [2].

3.1 Domain-Specific Service Interfaces

Web services are being increasingly used to implement e-business applications in various industry sectors. Travel industry examples include Galileo Web services (Galileo International, http://www.galileo.com/) [8], [23], Dollar Rent A Car (http://www.dollar.com/), and Southwest Airlines (http://www.southwest.com/) [14]. Interoperability of such industry domain-specific applications relies on well-defined service interfaces used consistently across the application domain. Adoption of standard service interfaces ensures that service providers (e.g. airlines) publish identical interfaces, avoiding the need to interpret the semantics of interfaces published by individual service providers. Standardized, domain-specific service interfaces should closely reflect business requirements for a particular application domain and in effect provide application developers with a high-level API for building domain-specific applications. APIs are used extensively in programming environments and have been recently applied to the problem of application interoperability. For example, the Open Knowledge Initiative (OKI) consortium is developing a specification for educational services in the form of a Java API to enable interoperability for learning technology platforms [26]. The OKI Open Service Interface Definitions (OSIDs) API is designed to provide separation, encapsulation, and layering to facilitate application interoperability and integration [25].

4. SERVICE INTERFACE DESIGN

Unlike the document-centric approach which is concerned with the design of standard documents, the service-centric approach focuses on the design of service interfaces. As noted earlier, well-designed service interfaces are a key requirement for interoperability in service-oriented e-business applications. In this section we first discuss related work on Web services design (section 4.1), and then describe a service interface design method based on identifying elementary business function and converting standard message formats into a set of corresponding service-interfaces (section 4.2). We then apply data engineering principles to refine the interface design, and show that normalization of interface parameters can lead to minimization of coupling and maximization of cohesion of service operations.

4.1 Related Work

Web services design is an active area of research with most existing approaches focusing on designing service interfaces from existing components and using object-oriented methods or component-based techniques [1], [12], [18], [24], [32]. For example, Ambler [1] proposed a method for deriving Web services from UML models. The method involves identifying class contracts that define public interfaces for a given class, and combining the contracts to reduce the number of services resulting in a cohesive collection of classes called domain packages. Papazoglou and Yang [18] describe a design methodology for Web services and business processes, defining business processes as sets of collaborative Web services. The methodology provides service design guidelines based on the principles of minimizing coupling and maximizing cohesion to

ensure that the resulting services are self-contained, modular, extendable and reusable, and produces definition of WSDL Web service interfaces and WSFL service flow models. The methodology also covers non-functional service design guidelines including service provisioning strategies and service policy management models. Stevens [24] focuses on the problem of designing Web services operations with appropriate level of granularity, differentiating between coarse-grained and fine-grained services based on the scope of functionality covered by the service. Wieringa, et. al. [32] describes design guidelines that rely on functional decomposition to produce modular Web services architecture. In summary, while there is some agreement about the basic interface design principles there are no widely used design frameworks for Web services design [6].

Rather then considering the design of Web services for individual enterprise applications or components we focus on the problem of defining domain-specific service interfaces (e.g. for the travel domain). We do not consider work flow aspects of Web services design and focus entirely on the design of service interfaces from a *programmatic* view point, adapting principles of API design and applying data engineering principles to refine the design of interfaces.

4.2 Design Framework for Domain-Specific Service Interfaces

The task of designing standard domain-specific service interfaces is conceptually similar to the design of application programming interfaces and we can draw from the extensive literature on this topic to identify guiding principles for interface design [16], [9], [11], [30], [28]. We summarize the key interface design principles below:

- Orthogonality – the functionality of interfaces should not overlap; each interface defines a distinct function within the application domain
- Completeness – interfaces should fully cover functionality of the application domain
- Minimality – interfaces should only be designed for common functions; specialized functions should be supported via an extendibility mechanism
- Universality – all service providers implement the standard set of interfaces
- Extendibility support via non-standard interfaces - service providers can define additional (non-standard) service interfaces to support specialized functions
- Evolution support via interface versioning - evolution should be supported via interface versioning, so that existing interfaces can be maintained to support *legacy* applications
- Clarity and elegance – interfaces should have well-defined semantics and should be easy to understand
- Uniformity – uniformity of naming services, operations, parameters etc.

These general principles are useful as guidelines for the design of service interfaces for a specific application domain. At the more detail level of interface design for individual services two design principles are of particular importance: maximizing method (i.e. Web services operation) cohesion and minimizing method coupling. Applying these principles to service interface design leads to improved clarity of the interfaces, reduction in undesirable side effects, and improved flexibility of applications [18], [29], [30].

In the following sections we describe a service interface design approach based on mapping elementary business functions to Web services operations, and applying the above interfaces design principles to maximize cohesion and minimize coupling of the resulting service interfaces. The OTA message specifications and accompanying description of business processes represent a comprehensive model for the travel application domain, and provide a useful starting point for the design process illustrated here. The design methodology presented in this paper is of general applicability and can be used to transform any document-centric specifications (e.g. EDI document definitions) into well-designed service interface definitions.

4.2.1 Identifying Operations

Service interface definition involves specification of operations including identification of input and output parameters for each operation. This task is analogous to designing method signatures in the context of object-oriented design, and requires that suitable candidate methods are identified. Our approach is based on decomposition of complex business functions into elementary business functions, i.e. simple atomic functions that cannot be further decomposed [5], [10]. We then map the elementary business functions to *simple* Web services operations and identify input and output parameters using the corresponding OTA message structures. This approach is consistent with maximizing method cohesion as elementary business functions typically accomplish a single conceptual task [30].

Now returning to our travel example introduced in section 2, we model the interaction between a travel agent and an airline using a Sequence Diagram as illustrated in Figure 2. For the purpose of this example we assume that a flight booking is for a single flight segment, i.e. a direct flight between the origin and destination locations (this corresponds to the Direct Flight Only option in the OTA AirAvailRQ message in Figure 1).

Each step in the dialog between a travel agent and an airline is modeled with a Request/Response message pair and corresponds to an elementary business function; consequently the granularity of operations is determined by the corresponding elementary business functions.

The following candidate operations can be identified from the Flight Booking Process Sequence Diagram in Figure 2 (brief descriptions of the business processes are loosely based on the OTA specification):

CheckFlight: The travel agent requests a list of flights for a pair of origin and destination cities on a given departure date. The airline responds with a collection of flights (i.e. zero or more flights). The response includes flight number, departure airport, departure time, arrival date, arrival time, and arrival airport for each flight.

CheckAvailability: The travel agent requests seat availability for a particular flight specifying the flight number, departure airport, arrival airport, departure date, and cabin type (e.g. economy). The airline responds with the quantity of seats available.

CheckPrice: The travel agent requests pricing information specifying the flight number, departure airport, arrival airport, departure date, and cabin type. The airline responds with the pricing information that includes the base fare and base fare code.

SeatingRequest: The travel agents request a traveler seating preference (e.g. an aisle seat). The airline responds with a seat number.

GetItinerary: The agent requests the travel itinerary specifying the booking reference number. The airline responds with itinerary information that includes traveler information, flight number, departure airport, arrival airport, departure date, departure time, arrival date, arrival time, cabin type, booking status, and journey duration.

More complex operations can be constructed by composition from the above defined low-granularity, elementary operations. For example, a flight between origin and destination locations (e.g. Sydney and London) in general consists of a number of flight segments (e.g. Sydney to Singapore, and Singapore to London) so that the corresponding flight availability request needs to check availability for each segment separately and provide programming logic to determine if the entire flight is available. Another example involves the travel agent requesting flight availability information from a number of airlines before making a booking decision based on some criteria (e.g. the lowest price).

4.2.2 Refining Interface Design

Following the identification of operations in section 4.2.1 above, we can define interfaces (input and output parameters) for each candidate operation using the OTA message structures as shown in Table 1.

An important interface design goal is to minimize inter-dependencies between applications. This requirement can be expressed as minimization of method coupling as formulated for design of methods in object-oriented programming [29], [30]. Minimization of coupling involves defining input and output parameters so that inter-dependencies and side-effects are minimized. This leads to consideration of data properties of parameters, and the general rule that only data that is used directly by a given method should be exposed as parameters to maximize encapsulation and minimize coupling.

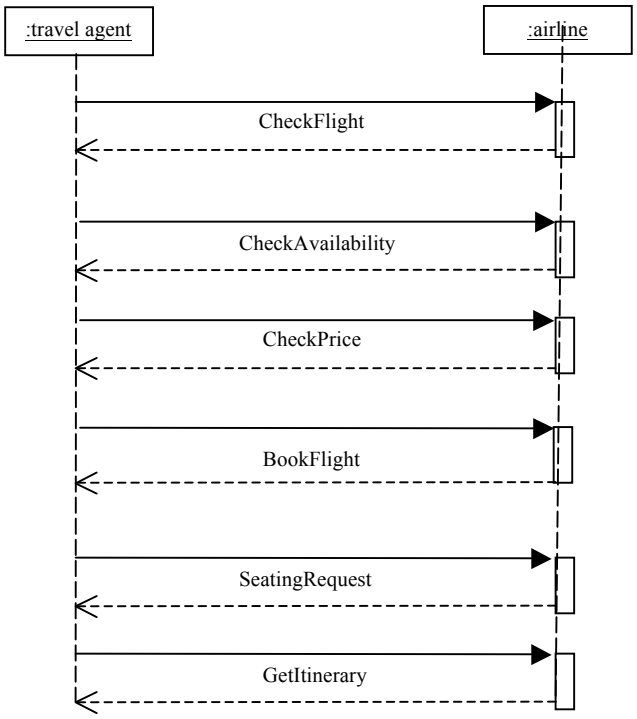

Figure 2. Flight Booking Process Sequence diagram.

All input parameters must be used by the method *as data*, i.e. not to control the execution of the method. Further, both input and output parameter sets should be *minimal;* this leads to the formulation the following interface design rules:

Rule i) input parameters should form a minimal set

Rule ii) output parameters should form a minimal set

Minimality implies that parameters are mutually independent; i.e. cannot be derived from each other based on functional dependencies. Venners [30] classifies methods according to the type of the request performed into three types: state-view methods (query operations that return data in output parameters, given query formulated using input parameters), state-change (methods that result in update, insert, and delete transactions based on input parameters), and utility methods (notifications, etc. methods that do not use data parameters). We formulate an additional rule for state-view methods to maximize cohesion:

Rule iii) output parameters must be fully functionally dependent on the input parameter set

Put in other words, method output should not include parameters that are not directly generated by the method from the input parameter set. In effect, the parameters of a State-View method form a relation where the input parameters are the key attributes and output parameters are non-key attributes, and therefore data normalization rules can be applied to this situation to ensure that output parameters are fully functionally dependent on the input parameter set [3]. State-change methods use values supplied via input parameters to create new records (i.e. insert records) or change existing records (i.e. update or delete existing records), and typically return a value that identifies the new record (e.g. BookingReferenceID, when booking a flight), or an

acknowledgement (e.g. when ordering a special meal). We now apply the above normalization rules to the interfaces of candidate operations in Table 1, assuming the following functional dependencies:

FD1: OriginLocation, DestinationLocation, DepartureDate

➔ FlightNumber

FD2: FlightNumber

➔ DepartureAirport, DepartureTime, ArrivalAirport, ArrivalTime

FD3: FlightNumber, DepartureDate

➔ ArrivalDate

FD4: FlightNumber, DepartureDate, CabinType

FD5: FlightNumber, DepartureDate, CabinType

➔ BasicFareCode, BasicFare

FD6: BookingReferenceID

➔ TravelerName, FlightNumber, DepartureAirport ArrivalTime, DepartureDate, DepartureTime, ArrivalAirport, ArrivalDate, ArrivalTime, CabinType, BookingStatus, JourneyDuration

Table 1. List of candidate operations and corresponding parameters for the Flight Booking Service

Operations	Input Parameters	Output Parameters
CheckFlight Method type: state-view	OriginLocation DestinationLocation DepartureDate	FlightNumber DepartureAirport DepartureTime ArrivalAirport ArrivalDate ArrivalTime
CheckAvailability Method type: state-view	FlightNumber DepartureAirport ArrivalAirport DepartureDate CabinType	Quantity
CheckPrice Method type: state-view	FlightNumber DepartureAirport ArrivalAirport DepartureDate CabinType	FareBasisCode BaseFare
BookFlight Method type: state-change	FlightNumber DepartureAirport DepartureDate TravelerName CabinType	BookingReferenceID
SeatingRequest Method type: state-change	BookingReferenceID SeatPreference	SeatNumber
GetItinerary Method type: state-view	BookingReferenceID	TravelerName FlightNumber DepartureAirport DepartureDate DepartureTime ArrivalAirport ArrivalDate ArrivalTime CabinType BookingStatus JourneyDuration

Applying Rule i) and using FD2 we can eliminate the parameters DepartureAirport and ArrivalAirport from the input parameter sets of CheckAvailability, CheckPrice, and BookFlight operations as these parameters can be derived from FlightNumber. Applying Rule ii) and FD2 we eliminate DepartureAirport, DepartureTime, ArrivalAirport, and ArrivalTime from output parameters of operation CheckFlight. This leaves FlightNumber and ArrivalDate in the output parameter set; but this violates Rule iii) as ArrivalDate

is partially dependent on input parameter DepartureDate. This leads to the elimination of ArrivalDate from the output parameter set. The resulting set of operations (Table 2) includes two new operations: CheckSchedule, and CheckArrival to preserve functional dependencies, and represents a set of operations with maximum cohesion and minimum coupling, i.e. consistent with the rules defined in this section. The trade-off for maximizing cohesion of and minimizing coupling of operations is an increased number of operations. The designer may choose to optimize the design by combining operations. For example, CheckAvailability and CheckPrice operations could be combined without violating the interface design rules as both have identical input parameters.

4.3 WSDL Specification

The resulting design of service interfaces for the Airline Booking Service given in Table 2 can be implemented in the form of WSDL specifications as described in our earlier publication [7]. The implementation of operations uses Web services RPC binding style (indicated in Table 2), consistent with the programmatic approach advocated in this paper.

Table 2. List of operations and corresponding parameters for Flight Booking Service

Operations	Input Parameters	Output Parameters
CheckFlight Method type: state-view Implementation: RPC	OriginLocation DestinationLocation DepartureDate	FlightNumber
CheckSchedule Method type: state-view Implementation: RPC	FlightNumber	DepartureAirport DepartureTime ArrivalAirport ArrivalTime
CheckArrival Method type: state-view Implementation: RPC	FlightNumber DepartureDate	ArrivalDate
CheckAvailability Method type: state-view Implementation: RPC	FlightNumber DepartureDate CabinType	Quantity
CheckPrice Method type: state-view Implementation: RPC	FlightNumber DepartureDate CabinType	FareBasisCode BaseFare
BookFlight Method type: state-change Implementation: RPC	FlightNumber DepartureDate TravelerName CabinType	BookingReferenceID
SeatingRequest Method type: state-change Implementation: RPC	BookingReferenceID SeatPreference	SeatNumber
GetItinerary Method type: state-view Implementation: RPC	BookingReferenceID	TravelerName FlightNumber DepartureAirport DepartureDate DepartureTime ArrivalAirport ArrivalDate ArrivalTime CabinType BookingStatus JourneyDuration

5. CONCLUSIONS

In this paper we have argued the benefits of the service-centric interoperability model for e-business applications, and we have presented a design method for domain-specific service interfaces that relies on the principles of maximizing method cohesion and minimizing method coupling. Using a Flight Booking example we have illustrated that data normalization when applied to the design of service interfaces can lead to

minimization of method coupling by avoiding the externalization of redundant information.

Using the proposed design framework for Web service interfaces in *real-world* e-business applications leads to an increased number of operations for a given Web service and consequently to a corresponding increase in the number of RPC calls required to implement a specific business function. This represents a challenge given the current Internet environment characterized by unreliable network connectivity and unpredictable response times, making the programmatic approach using low-granularity operations advocated in this paper only suitable for fast and reliable Intranet environments. However, given the rapid development of Internet technologies it is likely that Internet will provide service levels comparable to today's Intranet environments in the not too distant future, and this will make the programmatic approach viable for Internet-based e-business applications.

5.1 Service Granularity

Finding an optimal level of granularity for Web services and individual service operations requires further investigation. Coarse-grained operations tend to lack cohesion and limit design flexibility, while fine-grained operations increase the number of service interfaces and the number of RPC calls. It is possible to increase the granularity by combining operations based on common parameters as noted in section 4.2.2 for the CheckAvailability and CheckPrice operations. Combining operation can lead to loss of cohesion as the resulting operation no longer implements a single conceptual task, but this trade-off may well be justified given the benefits of reduced complexity of the application and reduction in RPC calls. The application of data engineering principles to service interface design has a potential to improve our understanding of the impact of increasing granularity of operations on cohesion and coupling. The precise method for combining elementary operations into larger granularity operations while minimizing undesirable side effects associated with loss of cohesion and increase in coupling warrants further research.

6. REFERENCES

[1] Ambler, S.W. *Deriving Web services from UML models, Part 1: Establishing the process*. http://www-106.ibm.com/developerworks/webservices/library/ws-uml1/, March 1, 2002.

[2] Bieber, G. and Carpenter, J. *Introduction to Service-Oriented Programming (Rev 2.1)*. http://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf, 2001.

[3] Codd, E.F. Normalized Data Structure: A Brief Tutorial. In *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control* (San Diego, California, November 11-12, 1971). ACM, 1971, 1-17.

[4] ebXML. http://www.ebxml.org, 2004.

[5] Eriksson, H.-E. and Penker, M. *Business Modeling with UML: Business Patterns at Work*. John Wiley & Sons, New York , USA, 2000.

[6] Feuerlicht, G., Meesathit, S. Design Framework for Domain-specific Service Interfaces. In *Proceedings of the 2nd International Workshop on Web Services: Modeling, Architecture, and Infrastructure, WSMAI-2004,* (Porto, Portugal, April 13 - 14, 2004). INSTICC Press, 2004, 109-115.

[7] Feuerlicht, G. Implementing Service Interfaces for e-business Applications. In *Proceedings of the second workshop on e-Business, WeB 2003,* (Seattle, USA, December 13-14, 2003). 2003.

[8] Fontana, J. Galileo travels down Web services path. *Network World [Online]*, April 9, 2002. http://www.nwfusion.com/news/2002/0429galileo.html,

[9] HKNET5. *API Design in Software Architecture*. http://hknet.tm.tue.nl/section33/tech_design.html, 2003.

[10] Larman, C. *Applying UML and Patterns : an Introduction to Object-Oriented Analysis and Design and the Unified Process* (2nd edn). Prentice Hall, Upper Saddle River, NJ, USA, 2001.

[11] Manageability Inc. *Principles of Loosely Coupled API's*. http://www.manageability.org/blog/archive/20030628%23principles_of_loosely_coupled_api/view, July 30, 2003.

[12] Mecella, M. and Pernici, B. Designing wrapper components for e-services in integrating heterogeneous systems. *VLDB Journal*, *10*, 1(2001), 2-15.

[13] Medjahed, B., Benatallah, B., Bouguettaya, A., Ngu, A.H.H. and Elmagarmid, A.K. Business-to-business interactions: Issues and enabling technologies. *VLDB Journal*, *12*, 1(May 2003), 59 - 85.

[14] Metz, C. Testing the Waters. *PC Magazine [Online]*, November 13, 2001. http://www.pcmag.com/article2/0,4149,154693,00.asp.

[15] Microsoft. *BizTalk*. http://www.microsoft.com/biztalk/, 2004.

[16] OpenOffice.org. *OpenOffice.org API-Design-Guidelines*. http://api.openoffice.org/docs/DevelopersGuide/Appendix/IDLDesignGuide/IDLDesignGuide.htm, 2004.

[17] OTA. *The Open Travel Alliance website*. http://www.opentravel.org/, 2004.

[18] Papazoglou, M.P. and Yang, J. Design methodology for Web services and business processes. In *Proceedings of the 3rd VLDB-TES workshop* (Hong Kong, August, 2002). Springer, 2002, 54-64.

[19] Piccinelli, G., Emmerich, W., Zirpins, C. and Schutt, K. Web service interfaces for inter-organisational business processes an infrastructure for automated reconciliation. In *Proceedings of the sixth international conference on Enterprise Distributed Object Computing (EDOC '02)* (Lausanne, Switzerland, September 17-20, 2002). IEEE, 2002, 285-292.

[20] Piccinelli, G., Salle, M. and Zirpins, C. Service-oriented modelling for e-business applications components. In *Proceedings of the Tenth IEEE international workshops on Enabling technologies: Infrastructure for collaborative enterprises (WET ICE 2001)* (Cambridge, MA, USA, June 20-22, 2001). IEEE, 2001, 12 - 17.

[21] RosettaNet. http://www.rosettanet.org/, 2004.

[22] Sayal, M., Sahai, A., Machiraju, V. and Casati, F. Semantic analysis of e-business operations. *Journal of Network and Systems Management*, *11*, 1(Mar. 2003), 13-37.

[23] Schwartz, E. Triple A Launches Web Service Airline Reservation System. *InfoWorld [Online]*, August 12, 2002.

http://www.infoworld.com/article/02/08/12/020812hntriplea_1.html,

[24] Stevens, M. *Multi-Grained Services*. http://www.developer.com/design/article.php/1142661, May 21, 2002.

[25] The Open Knowledge Initiative. *O.K.I. Architectural Concepts*. http://prdownloads.sourceforge.net/okiproject/OkiArchitecturalConcepts.pdf?download, December 5, 2003.

[26] The Open Knowledge Initiative. http://web.mit.edu/oki/, 2004.

[27] UNECE. *UN/EDIFACT (United Nations Directories for Electronic Data Interchange for Administration, Commerce and Transport)*. http://www.unece.org/trade/untdid/welcome.htm, 2004.

[28] Vallee-Rai, R. *Sable API Design Guidelines*. http://www.sable.mcgill.ca/publications/technotes/sable-tn-1998-1.ps, March 3, 1998.

[29] Venners, B. *API Design: The Object*. http://www.artima.com/apidesign/object.html, April 26, 2002.

[30] Venners, B. *Introduction to Design Techniques*. Note http://www.javaworld.com/javaworld/jw-02-1998/jw-02-techniques.html, February, 1998.

[31] W3C. *W3C Web Services Activity*. http://www.w3.org/2002/ws/, 2004.

[32] Wieringa, R.J., Blanken, H.M., Fokkinga, M.M. and Grefen, P.W.P.J. Aligning application architecture to the business context. In *Preceedings of 15th international conference on Advanced information systems engineering (CAiSE 2003)* (Klagenfurt, Austria, June 16-18, 2003). Springer, 2003, 209-225.