

# An Event Based Approach to Web Service Design and Interaction

Wilfried Lemahieu, Monique Snoeck, Cindy Michiels and Frank Goethals

Katholieke Universiteit Leuven, Department of Applied Economic Sciences,

Naamsestraat 69, B-3000 Leuven, Belgium

{wilfried.lemahieu, monique.snoeck, cindy.michiels, frank.goethals}@econ.kuleuven.ac.be

**Abstract.** This paper advocates an approach to web service design and interaction that is based on web services simultaneously participating in shared *business events*. In contrast to one-to-one method invocations, such events are *broadcast in parallel* to all web services that participate in it. Moreover, the transactional business events are distinguished from non-transactional *attribute inspections*. The paper first discusses the role of the business event concept as the cornerstone for a methodical analysis and design phase. Then, it is shown how the event broadcasting paradigm can be implemented by means of SOAP messaging.

## 1 Introduction

The advent of the web services paradigm brought about a revolution in the way business partners can integrate their information systems and allows for innovative organizational forms that were unthinkable before. The enabling technology is SOAP [1], the current de facto standard for web service interaction, which is based on the exchange of XML messages or documents.

Initially, XML based business-to-business interaction was mainly accomplished by replacing paper documents such as purchase orders, receipts, invoices etc. by an equivalent electronic document. XML was utterly useful as a data format for this purpose, as it allowed defining structured documents, which could be processed much more efficiently by applications than plain text documents. Moreover, the structure of a certain document type could be laid down in a DTD or an XML schema, such that a document's validity could be checked against its type definition. In this way, interacting companies could mutually agree on a common structure for each document type involved in their transactions. However, instead of being

document-based, automated business-to-business interaction can be accomplished by means of another paradigm as well. The latter originates in the concept of distributed object architectures such as DCOM [2], RMI [3] and CORBA [4], where interaction consists of objects on different hosts remotely invoking one another's methods. DCOM, RMI and CORBA are well established as *intra enterprise* application integration technologies in a local area network, however they can hardly be used for *inter enterprise application integration* over the Internet because they are too heavyweight and above all don't cope well with firewalls. SOAP, on the other hand, allows for remote method invocations and their parameters to be represented as *XML messages*, which are transmitted over "native" Web protocols such as HTTP or SMTP. It can be seen as a very lightweight RPC (Remote Procedure Call) mechanism over the Web, which deals well with firewalls.

Although SOAP supports document-based as well as RPC-based interaction, the latter seems to have the potential to become the predominant technology for B2B communication and inter enterprise application integration. Web services can then be defined as self-contained software components that expose specific business functionality on the Internet, such that other applications can use them by means of established web protocols and data formats such as HTTP and XML. Enterprises publish only a limited set of services instead of entire applications. This is very similar to the distributed object paradigm, where only a limited set of methods, specified in an object's public interface, can be called remotely, with the actual implementation details being *hidden* from the outside world. SOAP and the web services paradigm allow for heterogeneous information systems to be coupled in a very loose way, instead of having to resort to inflexible, tightly coupled infrastructures, which can hardly be used beyond a single company's walls. In this way, the integration of the business processes among multiple business partners can be supported by integrated information processing across all participating companies [5].

Although the web service concept looks very promising, it hasn't fulfilled its full potential yet. Current implementations are largely limited to rather simple request/response services such as currency converters, stock information services etc. More advanced transactional systems, where services belonging to multiple

business partners participate in complex business transactions, are still very scarce. One of the primary reasons is that standards for web service *transactions*, *composition* and *choreography* are still under development. Existing attempts such as BTP [6], WSCL [7], BPEL4WS [8] (in its turn successor to XLANG and WSFL) or BPML [9] are based on quite divergent assumptions. However, we think that another very important reason, which may also hamper the efforts with respect to web service choreography standards, is the fact that XML based RPC (and also XML document based interaction for that matter) is essentially a *one-to-one mechanism*: web service A invokes a method on service B, after which B may or may not send a return value to A. Such mechanism is indeed suitable for simple request/response services, but is inherently difficult to co-ordinate in a complex environment where numerous business partners interact in shared business processes. One-to-one messaging as a mechanism to co-ordinate processes that are shared by many parties inevitably results in intricate sequences of message exchanges, over which it becomes nearly impossible to co-ordinate transaction management. Also, whereas traditional LAN/WAN information systems based on distributed object/component architectures such as CORBA, DCOM or RMI/EJB typically resulted from a rigid analysis and design phase, this is often not the case with regard to web services. Frequently, individual services providing some functionality are published without a thorough analysis regarding their position within the business processes and the global information system architecture. Again, such approach is only suitable to isolated request/response services [10].

Therefore, this paper advocates an *event based* approach to web service development. Interaction between services is based on the simultaneous participation in (and processing of) *shared business events*. Such transactional business events are distinguished from (non-transactional) *attribute inspections*. Moreover, events are not propagated one-to-one but are *broadcast* in parallel to all services that have an interest in an event of the corresponding type. The remainder of the paper is structured as follows: in section 2, the concept of business events is used as the basis for a methodical analysis and design phase. Section 3 describes an *implementation* approach for event based interaction between web services by means

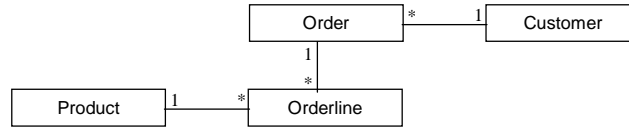
of SOAP messaging. Particular emphasis is put on the differences with event based interaction in a standalone or LAN environment. Section 4 formulates conclusions and discusses topics for future research.

## 2 Event based analysis and design

### 2.1 Business objects participate in shared business events

The event based interaction mechanism as proposed in this paper directly reflects an underlying object-oriented analysis and design methodology: MERODE [11; 12]. This method represents an information system through the definition of business events, their effect on enterprise objects and the related business rules. Although it follows an object-oriented approach, it does not rely on message passing to model interaction between domain object classes as in classical approaches to object-oriented analysis [13; 14; 15]. Instead, *business events* are identified as independent concepts. An object-event table (OET) allows defining which types of objects are affected by which types of events. When an object type is involved in an event, a method is required to implement the effect of the event on instances of this type. Whenever an event actually occurs, it is broadcast to all involved domain object classes.

For example, let us assume that the domain model for an order handling system contains the four object types CUSTOMER, ORDER, ORDER LINE and PRODUCT. The corresponding UML [13; 16; 17] Class diagram is given in Fig. 1. It states that a customer can place 0 to many orders, each order being placed by exactly one customer. Orders consist of 0 to many order lines, each order line referring to exactly one product. Products can appear on 0 to many order lines.



**Fig. 1.** Domain model for an order handling system

Business event types are e.g. `create_customer`, `modify_customer`, `create_order`, `ship_order`, etc. The object-event table (see Table 1) shows which object types are affected by which types of events and also indicates the type of involvement: C for creation, M for modification and E for terminating an object's life. For example, `create_orderline` creates a new occurrence of the class `ORDERLINE`, modifies an occurrence of the class `PRODUCT` because it requires adjustment of the stock-level of the ordered product, modifies the state of the order to which it belongs and modifies the state of the customer of the order. Notice that Table 1 shows a maximal number of object-event involvements. If we do not want to record a state change in the customer object when an order line is added to one of his/her orders, it suffices to simply remove the corresponding object-event participation in the object-event table. Full details of how to construct such an object-event table and validate it against the data model and the behavioral model is beyond the scope of this paper but can be found in [11; 12].

**Table 1.** Object-event table for the order handling system

	CUSTOMER	ORDER	ORDERLINE	PRODUCT
create_customer	C			
modify_customer	M			
end_customer	E			
create_order	M	C		
modify_order	M	M		
end_order	M	E		
customer_sign	M	M		
ship_order	M	M		
start_billing	M	M		
create_orderline	M	M	C	M
modify_orderline	M	M	M	M
end_orderline	M	M	E	M
create_product				C
modify_product				M
end_product				E

## 2.2 Modeling behavior

The event based data model is combined with a *behavioral model*. Indeed, the business objects are allowed to put *preconditions* on the events in which they participate. Some of these are based on *class invariants*, others on *event sequence constraints* that can be derived from a finite state machine associated with the object type. For example, when a customer orders a product, a new order line is created. In terms of enterprise modeling, this requires the following business events: *create\_order*, *modify\_order*, *create\_orderline*, *modify\_orderline* and *end\_orderline*. The *customer\_sign* event models the fact that a final agreement with the customer has been reached (signature of sales order form by the customer). At the same time this event signals that the order can be prepared for shipping. These kinds of sequence constraints can be modeled as part of the life-cycle of business objects. In the given example this would be in the life cycle of the sales order domain object. As long as it is not signed, a sales order stays in the state “existing”. The *customer\_sign* event moves the sales order into the state “registered”. From then on the sales order has the status of a contract with the customer and it cannot be modified any more. This means that the events *create\_orderline*, *modify\_orderline* and *end\_orderline* are no longer possible for this sales

order. The *ship\_order* event signals that the order has been shipped to the customer. As an effect of this event, the sales order object reaches the "shipped" state. Finally, the *start\_billing* event signals the start of the billing process. The resulting finite state machine is shown in Figure 2. In this way, the sequence constraints mimic the general business process.

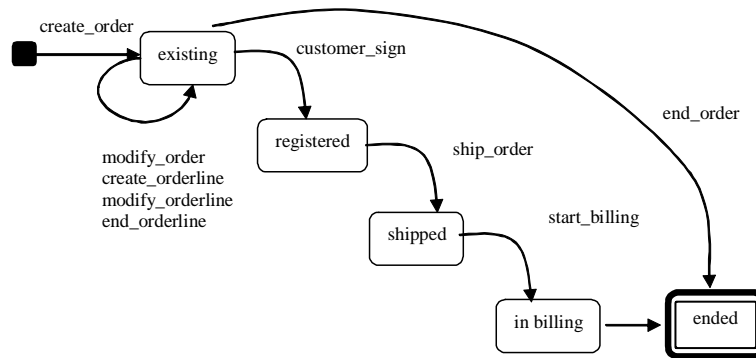


Fig. 2. State machine for a Sales Order object

The event concept as described in this section in the first place pertains to the *analysis* and *design* of an information system. The approach allows for the precise identification of an information system’s behavior, which is related to real world *business events*. Moreover, the constraints on this behavior can be accurately represented as preconditions imposed by individual business objects. In this way, the “choreography” of the interacting objects is distributed among the respective objects that participate in an event and the behavior of the entire system is the union of the individual objects’ behavior, which greatly reduces the complexity of behavioral modeling.

Another important advantage of this approach is that the results from an event-based analysis can be implemented by means of both object-oriented and non-object-oriented technologies. Currently, existing MERODE based implementations are built around e.g. *stored procedures* [12] or an *EJB framework* [18]. Moreover, even within an object oriented framework, the event based interaction paradigm (i.e. the specification as resulting from the analysis) can be *implemented* in multiple ways: by means of a

programming language that incorporates a native “event” construct but also by means of *method invocations* that “mimic” event propagation. In the latter case, an object type should have a method for each event type in which it may participate. Event propagation is “simulated” by an *event dispatcher* simultaneously invoking the appropriate method on each object instance that participates in the event. Hence the difference with traditional method based interaction lies in the fact that appropriate methods *are executed in parallel and in a co-ordinated way on all business objects that participate in the event.*

Previously, the focus of the MERODE methodology was upon implementations where the scope was a standalone system or a LAN based distributed object framework. Even if the applications themselves were distributed, there was always a common *enterprise layer* that contained the “primary copy” of all business objects [18]. The next section discusses how the same approach can also, *mutatis mutandis*, be applied in a Web environment. It is shown how web services may greatly benefit from an event based approach, at the analysis and design level as well as at the implementation level. In this respect, it is not our goal to come up with a competing mechanism to the current de facto standard web services stack of SOAP, WSDL [19] and UDDI [20]. The entire approach will be fit into this framework, i.e. events being dispatched as SOAP based “method invocations” over HTTP, similarly to the method based event dispatching mechanism mentioned above.

### **3 Application of the event paradigm to web service interaction**

#### **3.1 SOAP as a mechanism for event dispatching**

In general, SOAP is a *method based* interaction mechanism: SOAP messages represent remote procedure calls between web objects. Note that initially, we will discuss the interaction between “web objects” in general and use the terms “business objects” and “web services” intermittently. From section 3.3 onwards, we will explicitly distinguish between fine grained *business objects* and course grained *web services*. A SOAP based remote method invocation from object A to object B is accomplished in the following steps:

- A sends an XML message to B, with the method name and parameters as content.
- B receives the message and executes the corresponding method with the given parameters.
- As part of the execution of the method, some values of B's attributes may be updated.
- Optionally, B sends a return value to A.

Hence a SOAP based method invocation by A on B may have two effects: it may cause a state change in B, i.e. updates to one or more of B's attributes (as part of the *execution* of the method) and it may pass information from B to A (by means of its *return value*). Also, the interaction is essentially one-to-one: between the object that invokes the method and the object on which the method is invoked.

The *event* paradigm differs from the standard RPC approach as applied in SOAP in that the cases of A provoking a state change in B and A retrieving information from B are separated. With an event based approach, a firm distinction can be made between operations (as invoked by A) that "*read from*" attributes in B and that "*write to*" attributes in B. The latter is called the *Command-Query Separation* design principle, which, as discussed in [21], makes software much more transparent and makes classes easier to (re)use. Moreover, the event based paradigm also differs from the standard SOAP approach in that the "write" operation is essentially a *broadcasting* mechanism instead of a one-to-one method invocation: an event is broadcast simultaneously to (and processed in) all objects that participate in the event. However, as discussed above, an event based interaction paradigm may well be *implemented* by means of a method invocation mechanism.

### 3.2 Event broadcasts and one-to-one attribute inspections

With an event based approach, two generic types of interaction between web objects/services A and B can be discerned, which can both be implemented by means of SOAP messaging: *attribute inspection* and *event broadcasting*. These types are intertwined when a generic message based approach is used, which makes

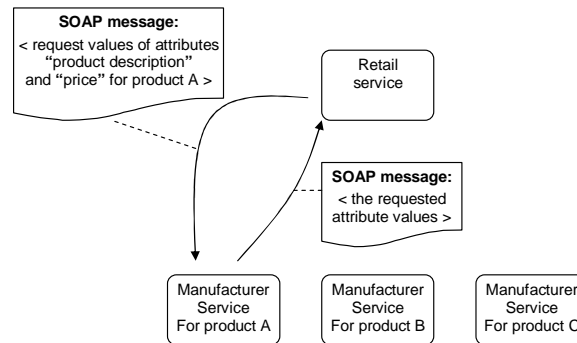
web service choreography and transaction modeling much more intricate. This section discusses how both attribute inspection and event broadcasting can be implemented as separate types of actions by means of SOAP messaging. Without burdening generality, for both types of interactions, we'll assume that object A takes the initiative.

**Attribute inspections.** Object A “reading” from B, i.e. A inspecting B’s attributes, is a type of interaction that does not cause a state change in B; B’s attributes are never “written to”. It occurs if object A “needs” information that is stored as attribute values (or as some query function) in B, e.g. because this information is used by object A to provide output to a human user or to test a precondition.

Attribute inspection is a rather straightforward type of interaction and is based on a one-to-one process. Only two objects are involved: the one needing the information (A) and the one that has the information as a value to one of its attributes (B). The interaction does not cause a state change on either of the two objects; therefore, it should not be the subject of transaction contexts. Also, preconditions resulting from class invariants or sequence constraints are not applicable to attribute inspections, again because they do not involve a state change<sup>1</sup>. SOAP messages that represent attribute inspections are very similar to calling a *getAttribute()* method on an entity Bean in the Enterprise JavaBeans (EJB) framework [22]. Applications or services can call directly upon published *getAttribute()* methods to retrieve data from an entity. Obviously, in a SOAP environment, performance can be boosted if multiple correlated attribute inspections are bundled into a single message exchange. Hence a SOAP message type should be defined for each type of (combined) attribute inspection and for the corresponding return value(s). Figure 3 presents the example of a web service implementing an online retail shop, which, when providing product information to its users, retrieves the detailed product specifications from the respective manufacturers.

---

<sup>1</sup> The sole exception may be checks for the appropriate access privileges, which is beyond the scope of this paper.



**Fig. 3.** Example of attribute inspections between web services

**Event dispatching.** The situation where object A “writes” to B, i.e. causes attribute values in B to be updated, is a bit more complex: because the updates that result from a given business event are to be coordinated throughout the entirety of all business objects that participate in the event (the combined updates can be considered as a single transaction), an object should never just *update* individual attributes of another object (in EJB terminology: *setAttribute()* methods should never be published in an entity Bean’s public interface). Changes to an object’s attributes are only to be induced by generating *business events* that affect the state of all business objects involved in the event. This type of interaction results from an event that occurs in the real world, e.g. a customer issuing a purchase order, a stock dropping below a threshold value, an order being shipped, ... This real world event affects the *state* of the real world. If this state change pertains to the part of the real world that is represented in the respective business object’s data, the state of the corresponding objects is to be updated accordingly. The respective business objects can subscribe to business events that are relevant to them (as denoted in the object event table). This relevancy can exist because their *state* is affected by the event and/or because they entail certain constraints that may decide whether or not the event is allowed to happen at all. These constraints are defined as *preconditions* on the corresponding event type.

This is accomplished by broadcasting an event object, which is an instance of a certain *event type* and has *attributes* that describe the event, to all business objects that participate in the event. As stated earlier, event broadcasting can be implemented by means of simultaneous method invocations on the object instances that participate in the event. The method invocations' parameters represent the event's attributes. In a SOAP context, the latter is implemented by means of an *event dispatcher* (cf. infra) sending SOAP messages simultaneously to all objects/services that participate in the event. Such message refers to the method that implements the object's reaction to the event and contains the event's attributes, which are passed as parameters to the invoked method. Hence there will be an XML message type for each event type that may occur and a web object will have a method that implements the object's reaction for each event type in which it may participate. If a relevant event occurs, each participating object receives a SOAP message and executes the corresponding method with the parameters provided in the message. Hence SOAP based event dispatching is still implemented by means of remote "method invocations". However, instead of one-to-one method invocations, they are simultaneous method invocations on several objects, with each method having the same name, as associated with a particular event type. These methods check constraints pertinent to the corresponding (object type, event type) combination and execute the necessary updates to attributes of that particular object if all constraints are satisfied. If not all constraints are satisfied, an exception is generated.

An example is presented in Figure 4. For instance a *create\_orderline* event is broadcast to three web services, each possibly putting constraints on the event and possibly changing its state because of the event:

- the online *Retail* service checks that the line number is unique and adds the orderline to the appropriate order
- the *Manufacturer* service checks whether the ordered product is in stock and adapts its stock level
- the *Customer* service checks whether the order's total bill does not exceed a certain amount

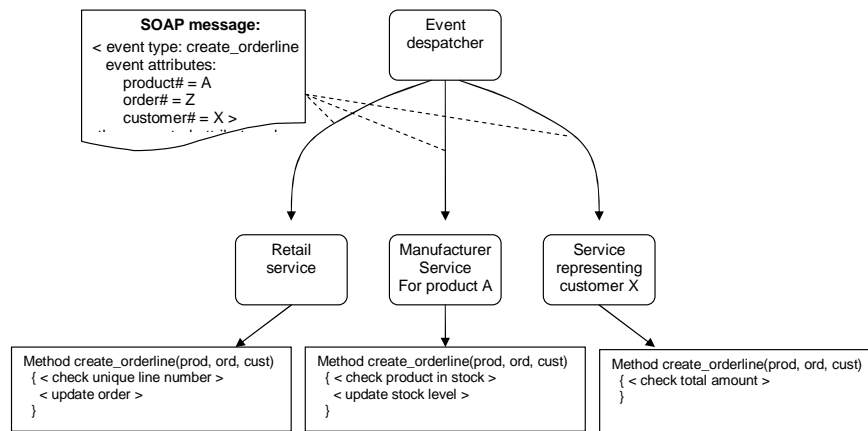


Fig. 4. Example of event dispatching between web services

The global result of the business event corresponds to the combined method executions in the individual web objects. The transaction is only committed if none of the objects that take part in the event have generated an exception. Otherwise, a rollback is induced<sup>2</sup>. Note that no return values are sent (as is the case with a method invocation), other than possibly a status variable for transaction management purposes, i.e. to determine whether the transaction can be committed or should be rolled back.

### 3.3 Web services, local objects and remote objects

The interaction mechanism described so far indeed corresponds to event based interaction over SOAP. This interaction currently takes place between *business objects*, similar to the LAN based distributed object approaches such as DCOM, CORBA and RMI. However, an important difference between a Web based approach and a standalone or LAN based “enterprise layer” is that the business objects and the business logic of the interacting organizations are to be grouped into coarser grained components: the actual *web services*. Indeed, it would not be feasible for a company to publish individual interfaces to its actual

<sup>2</sup> Or, instead of a global rollback, corrective actions are taken. A “hard” commit-or-rollback strategy is often too limiting in a web services context with long-standing transactions.

business objects. Rather, it will publish one or more *web services* that fulfill an actual business function and that are composed of (and affect) multiple internal business objects.

Moreover, the enterprise layer, i.e. the entirety of business objects that shapes the business process(es), is now itself distributed across different sites/services, with each service controlling/consisting of a subset of the business objects. Therefore, from a given service's perspective, we can distinguish between *local* objects (the ones that make out the service) and *remote* objects (the ones that are part of other services, possibly on other sites). A single service can be considered as the unified *public interface* of the underlying business objects, which allows for events to be induced and attributes to be inspected. The entirety of business objects that implements a single service should be accessible through a single *port*: this "flat" web service interface *hides* the underlying complexity of the business objects from other services, much like an individual object's implementation is hidden from the other (local or remote) objects it interacts with. Hence a given object should only be aware of the other local objects; a given service should only be aware of (and interact with) other services, not with these services' constituting business objects. The actual behavior of the entire service is then the combined behavior of the business objects it encompasses. However, differently to the "traditional" web service approach, its operations represent event broadcasts and attribute inspections, instead of all-purpose remote procedure calls.

### **3.4 Event based interaction in two stages and explicit subscription to remote events**

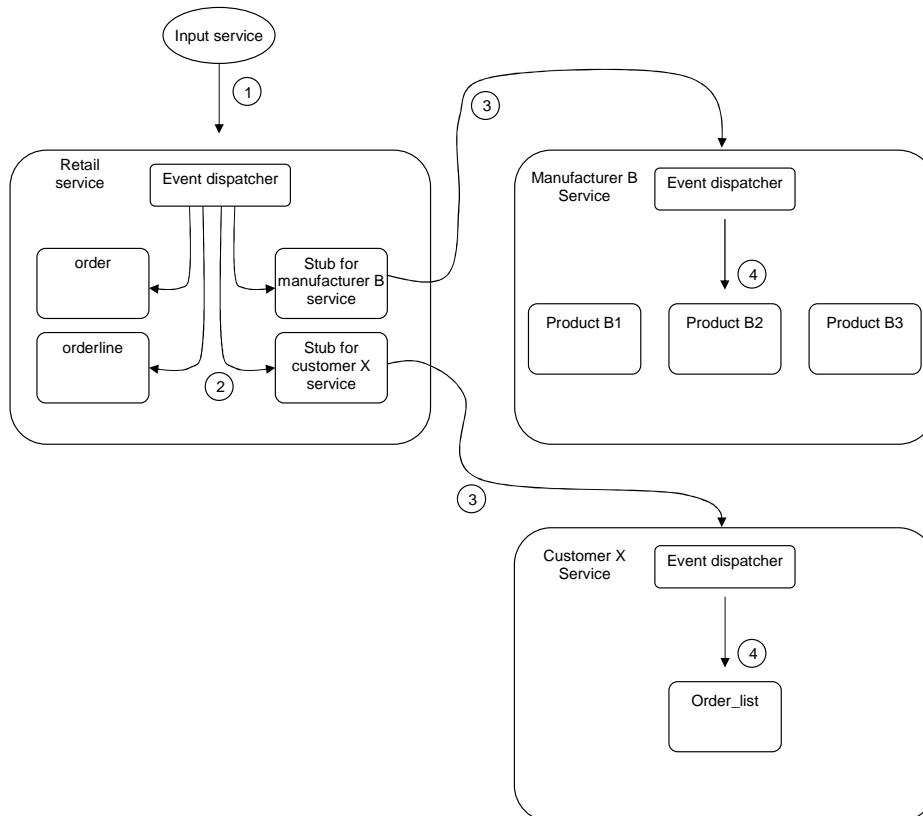
In a LAN environment, event broadcasting can be implemented by means of a single, central event dispatcher. "Real world" events are translated into business events in the information system by means of so-called *input services*. In general, these will encompass user interface components that receive input from human beings about relevant real world events, e.g. the "sign" button in a sales order form. In some cases, real world events may be detected without human intervention, e.g. a sensor that detects when the stock of a liquid product drops below a certain threshold. An input service translates a real world event into an event object, which is broadcast by the event dispatcher. The latter "knows" which event types are

relevant to which object types; this information can be encoded, based on the object event table that resulted from the event based analysis.

The same principle can be applied in a web services environment. Now each web service may have its own local input services, which generate local business events. These events can be dispatched to the service's local objects that participate in the event, based on a "local" object event table. However, the assumption of a single event dispatcher that also automatically knows all *remote* objects to which a certain event may be of interest is unrealistic: in many cases, web services are developed without prior knowledge of the other services they are to interact with, and certainly without knowledge about these services' internal business objects. Therefore, as to remote objects to which the event may be relevant, the approach should cater for an explicit *subscription* mechanism. In this way, a given web service's event dispatcher will dispatch its events to all local objects *and* to all remote objects that are explicitly subscribed to the corresponding event type.

However, because the basic principles underlying web services involve a level of *information hiding* at the service level, i.e. other services only get to see a service's interface and not its implementation details, it would not be a good idea for event propagation to take place between the *business objects* that make out the different services. As already said, a service should only be aware of its peer services, not of their constituting objects. Therefore, the (remote) event propagation mechanism is implemented at the *service* level: an event is propagated to all services to which the event is relevant. If one of a service's objects should be aware of a certain remote event type, the entire *service* subscribes to this event. Subscription of a remote service comes down to a reference to it being stored as an attribute value to one of the local service's objects. For that purpose, a web service contains *stub objects* that locally "represent" an individual remote service and that contain a reference to its URL. The resulting event based interaction mechanism takes place in four stages: if an event occurs in a given web service (as initiated by one of its local input services), this event is broadcast to all appropriate *local* objects, among which the stub objects. Each stub object propagates the event to the external service it represents. Such remote service then in its

turn broadcasts the event to its own local objects. The example from Figure 4 is now elaborated upon in Figure 5. First, the input service associated with the online retail service detects a real world event (1), which corresponds to a request for the creation of a new orderline in an existing sales order. This event is broadcast by the retail service’s event dispatcher to all of the retail service’s *local* objects, among which some stub objects (2). The stubs for the corresponding manufacturer and customer propagate the event to the respective remote services they represent (3). Each remote service’s event dispatcher broadcasts the event to its own local objects (4). Figure 4 already depicted how the appropriate (local and remote) objects then each execute a corresponding method, in which preconditions are checked and/or updates are executed.



**Fig. 5.** Example of event propagation by means of stub objects

Hence the event based interaction mechanism can be applied on two different levels in the web service architecture: in the first place for interaction *between* web services, where each web service is perceived as an atomic unity by its peer services. The services will interact by responding to communal events. A second interaction mechanism should exist at the level of the *intra-service* interaction between the different business objects that make out a single service. Here, the reaction of the web service as a whole to a given event will be translated into the different business objects that make out the service reacting to the event. The same goes for the constraints: the constraints put on an event by a given service will be the combination of constraints put on that event by the individual objects that implement the service. The *alphabet* of event types “understood” by a service is the union of the alphabets understood by the individual objects that make out the service. If the same SOAP based interaction mechanism is applied to both inter-service and intra-service interaction, these objects can be considered as small “atomic” services themselves. This approach can be easily generalized into an n-level system: a service or component receives an event notification and propagates it to its constituting components, which in their turn propagate it to their components etc. In this way, the event is propagated recursively at each level in a hierarchy of complex web services with a complex task, that in their turn are conceived of more simple services with a simpler task until a level of atomic services is reached. On the other hand, at each level, a component that receives an event notification may be a “wrapper” that internally consists of components that interact by means of another mechanism. In this way, e.g. interaction at the service level may be event based, whereas some of the services consist of business objects implemented as Enterprise JavaBeans, which interact by means of RMI.

### **3.5 Creation of stub object instances and event subscription**

The event based interaction mechanism can be applied to web services that belong to long standing business partners with more or less established interaction patterns and to partners that participate in short lived, ad hoc partnerships. In the first case, one could start out from a “unified” analysis and design over

the extended enterprise, resulting in a “unified” business model that entails the business objects of all partners involved. In this approach each web service, from the moment it is deployed, has a stub object for each service it interacts with. As to ad hoc interaction, stub objects will be created at runtime, when a certain remote service is selected for interaction. Indeed, just like any object, a stub object will also be an instance of an object type. For example, an online retailer will have an object type for stubs representing remote “manufacturer” services and an object type for stubs representing remote “customer” services. The object event table discerns between the different effects a certain event type may have on a given object type: M(odify), C(reate) or E(nd). An event with a “create” effect on a stub object type will result in a new stub object representing a certain remote web service being created, at least if all constraints are satisfied. One of the event parameters should be the URL of the remote service, e.g. as retrieved from a UDDI repository. This URL is then stored as an attribute value to the stub object. From then on, the stub is able to send event messages to the remote service it represents and to retrieve attribute values from this service’s public attributes. In this way, one service *subscribes* to another service’s events. In a similar way, events with an “end” effect on a stub object terminate the interaction between two services.

## **4 Conclusions and future research**

### **4.1 Thorough analysis and design based on the “business event” concept**

Regardless of the implementation environment, it is of utter importance when developing an information system to start with a rigorous analysis and design phase. Due to their nature, the latter is often overlooked with respect to web services. In our approach, at the specification level, the explicit identification of business events has the advantage that constraints, representing the actual business logic, can be identified as preconditions for each (event type, object type) combination. The relation between static and dynamic specification is made by means of a very elegant and concise modeling technique: the object event table. Another principal design concept is *command-query separation*: the distinction between transactional event

broadcasts and non-transactional attribute inspections. A thorough analysis and design is key if one aims at complex transactional web service interaction, instead of simple request/response systems.

#### **4.2 Loose coupling, SOAP and WSDL compliance**

The event based specification can be implemented by means of event dispatching being simulated through simultaneous SOAP messages. Hence events can be used without giving up on the current de facto standard web services stack based on SOAP, WSDL and UDDI. Whereas the web service concept in itself already entails a loose coupling mechanism, the coupling between web services communicating by means of event broadcasting can be kept even looser: a web service does not communicate one-to-one with its peers but just broadcasts an event, which is processed by any service with an interest in it. In this way, e.g. the number of parties that participate in an event can be easily increased by just adding another service that subscribes to the event, without having to redesign the entire chain of one-to-one message exchanges.

In this respect, a future research topic is a standardized way of specifying an event based web service's interface by means of WSDL. Obviously, this is possible, as our event based mechanism was superimposed over SOAP messaging, which is the basis for WSDL. However, it would be a good idea to specify a generic framework for event based web service description (and discovery) by means of WSDL, where one explicitly discerns between event broadcasting and attribute inspection. In this way, a web service's interface can be specified as the combination of the event types it "understands" and the attributes it publishes externally.

#### **4.3 Web service composition, choreography and transaction management**

As already discussed in section 3.4, event propagation can be used both for interaction between peer services and for a complex service to co-ordinate the behavior of its components. In this respect, a future research topic is the optimization of the process of aggregating individual business objects (or "simple"

web services) into more complex services. Based on the event types “understood” by the respective business objects and the need for attribute inspections among business objects, the degree of *cohesion* between the respective objects can be calculated to determine an appropriate way to divide the objects across the actual services.

Moreover, the clear distinction between attribute inspections (which do not entail state changes) and business events (which do entail state changes) allows for focusing on only the latter with respect to *transaction management*. Also, the fact that event propagation is a broadcasting mechanism allows for more adequate web service transaction specification: a single business event results in multiple simultaneous updates in multiple business objects. The latter is much easier to describe than the myriad of one-to-one message exchanges that could make out a single business transaction in a pure RPC based approach. A topic for future research is a standardized transaction management mechanism, based on return messages sent by objects and services, which indicate whether or not any constraints were violated by the event in a certain object or service.

Although web service *choreography* can be situated as a layer above transaction management in the web services stack, there is a certain measure of overlap. Indeed, the reliability of physical inter-service communication channels may be much lower than the infrastructure that is offered in a LAN. Therefore, the interaction mechanism should allow for asynchronous interaction and specific transaction management mechanisms that are less strict than the “ACID” properties as we know them in a LAN environment. More specifically, “long-lived” transactions will not always be committed or rolled back in their entirety, but will consist of sub-transactions for which an alternative (i.e. a “corrective” action) has to be sought, if one of them fails. The latter already belongs to the layer of web service choreography. In this respect, the notion of events combines well with state machines, where events are the triggers for transitions from one state to another. This allows for the business process to be represented in the state machines of individual business objects and opens up great possibilities for the specification of web service choreography at various levels of aggregation.

#### 4.4 Events representing genuine business semantics

Finally, the event based approach to web services as proposed in this paper also in a certain way reconciles the two paradigms discussed in section 1. First, there is the *document based* paradigm, where the XML documents exchanged by means of SOAP actually represent electronic versions of purchase orders, receipts, invoices etc. These documents represent a true business reality and are easily understood by people with a business background. On the other hand, there is the RPC based approach, where the XML documents exchanged over SOAP actually consist of short messages, which represent (remote) *method invocations*, similar to what is done in e.g. DCOM, RMI or CORBA. Obviously, this RPC based approach is best understood by software developers. However, it does not really represent a business context. The event paradigm has the advantage of embodying both a business and a programming concept: business events such as purchases, out-of-stock events etc. but also events in the programming sense of the word, i.e. to which objects can *subscribe*. In the future, the business “meaning” of the events may also facilitate web service description at a *semantic* level by means of RDF [23], as advocated e.g. in [24].

## References

1. Seely, S., Sharkey, K.: SOAP: Cross Platform Web Services Development Using XML, Prentice Hall PTR, Upper Saddle River, NJ (2001)
2. Sessions, R.: Understanding COM and DCOM: Microsoft’s Vision for Distributed Objects, John Wiley & Sons, New York, NY (1997)
3. Pitt, E., McNiff, K.: Java.rmi: The Remote Method Invocation Guide, Addison-Wesley, Reading, MA (2001)
4. Siegel, J.: CORBA – fundamentals and programming, John Wiley & Sons, New York, NY (1996)
5. Leclerc, A.: Distributed Enterprise Architecture, Integrating the Enterprise, Vol. III, no. 5 (2000)
6. Potts, M., Cox, B., Pope, B.: Business Transaction Protocol Primer, OASIS Committee Supporting Document (2002)
7. Banerji, A. et al.: Web Services Conversation Language (WSCL) 1.0, W3C Note (2002)

- 22 Wilfried Lemahieu, Monique Snoeck, Cindy Michiels and Frank Goethals
8. Weerawana, S., Curbera, F.: Business Process with BPEL4WS, IBM white paper (2002)
  9. Arkin, A.: Business process Modeling Language, BPMI draft specification (2002)
  10. Frankel, D., Parodi, J.: Using Model-Driven Architecture to Develop Web Services, IONA Technologies white paper (2002)
  11. Snoeck, M., Dedene, G.: Existence Dependency: The key to semantic integrity between structural and behavioral aspects of object types, IEEE Transactions on Software Engineering, Vol 24 No. 24, pp.233-251 (1998)
  12. Snoeck, M., Dedene, G., Verhelst M, Depuydt A. M.: Object-oriented Enterprise Modeling with MERODE, Leuven University Press, Leuven (1999)
  13. Booch, G., Rumbaugh, J., Jacobson, I.: The unified modeling language user guide, Addison-Wesley, Reading, MA (1999)
  14. D'Souza, D. F., Wills, A. C.: Objects, Components and Frameworks with UML, The Catalysis Approach, Addison-Wesley, Reading, MA (1999)
  15. Jacobson, I., Christerson, M., Jonsson, P. et al.: Object-Oriented Software Engineering, A use Case Driven Approach, Addison-Wesley, Reading MA, Rev. 4th pr. (1997)
  16. Fowler, M., Kendall, S.: UML Distilled: applying the standard object modeling language, Addison-Wesley, Reading, MA (1998)
  17. Marshall, C.: Enterprise Modeling with UML: Designing Successful Software through Business Analysis, Addison-Wesley Professional, Reading MA (1999)
  18. Lemahieu, W., Snoeck, M., Michiels C.: An Enterprise Layer Based Approach to Application Service Integration, Business Process Management Journal, (forthcoming)
  19. Web Services Description Language (WSDL) 1.0. specification, <http://msdn.microsoft.com/xml/general/wsdl.asp> (2001)
  20. UDDI Technical White Paper, Ariba, IBM Corporation and Microsoft Corporation, (2000)
  21. Meyer, B.: Object-oriented software construction, second edition, Prentice Hall PTR, (1997)
  22. Matena, V., Stearns, B.: Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform, Sun Microsystems Press (2001)

23. Lassila, O., Swick, R. (Eds.): Resource Description Framework Model and Syntax Specification, W3C Recommendation (1999)
24. Lemahieu, W.: Web service description, advertising and discovery: WSDL and beyond, in: Vandenbulcke J. and Snoeck M. (eds.): New Directions In Software Engineering, Leuven University Press, Leuven (2001)