Event Based Web Service Description and Coordination

Wilfried Lemahieu, Monique Snoeck, Cindy Michiels, Frank Goethals, Guido Dedene, Jacques Vandenbulcke

Katholieke Universiteit Leuven, Department of Applied Economic Sciences,
Naamsestraat 69, B-3000 Leuven, Belgium
{wilfried.lemahieu, monique.snoeck, cindy.michiels, frank.goethals, guido.dedene, jacques.vandenbulcke}@econ.kuleuven.ac.be

Abstract. This paper proposes the concept of *business events* as the cornerstone to web service description and coordination. First, a web service architecture is introduced as the result of an event based analysis & design phase. Then, it is advocated how the event concept can be used for semantically rich web service description. A distinction is made between two web service interfaces: a non-transactional *query interface* and a transactional *event notification interface*. Furthermore, a web service *composition model* is proposed, based on *event broadcasting* and *event preconditions*, instead of traditional one-to-one method invocations. The composition model is presented in a static variant and in a version with dynamic subscription. Throughout the paper, it is shown how the event based approach fits entirely within the current standard SOAP/WSDL/UDDI web services stack.

1 Introduction

The web services concept can be considered as a revolutionary paradigm for loosely coupled application integration within and across enterprise boundaries. It promises to bring about a revolution in the way business partners can integrate their information systems, allowing for innovative organizational forms that were unthinkable before [1]. A web service can be looked upon as a *public, remote interface* to certain functionality, where the actual implementation is *hidden* from the applications that use it. In this aspect it is very similar to distributed object technologies such as RMI, CORBA and DCOM, which are, however, restricted to the intranet. In contrast, web services use a lightweight XML messaging protocol, SOAP [2], which is applicable across the entire Internet, without being hampered by companies' firewalls.

Still, despite its obvious great promises, the web service paradigm hasn't fully lived up to its expectations (yet), at least not at the *inter-enterprise* level. Indeed, web services are well established for intra-enterprise application integration (EAI) and even *static* business-to-business interaction (B2Bi), i.e. in an extended enterprise with fixed, long-standing business partners. However, they fail to provide more than very basic services at the level of *dynamic* B2Bi, where services dynamically find one another and enter ad-hoc partnerships to perform complex business transactions.

Current implementations of dynamic B2Bi are largely limited to rather simple request/response services such as currency converters, stock information services, i.e. non transactional systems. In this respect, web service technology didn't succeed in fulfilling two of its primary promises: fully automated *discovery* and *invocation* of (remote) services and fully automated *composition* of "atomic" services into ad-hoc complex, transactional systems.

This paper advocates how failure to achieve those two goals can, at least partially, be imputed to the one-to-one interaction paradigm that inherently underlies "traditional" SOAP messaging. As an alternative, an approach to web service interaction is proposed, based on the simultaneous participation in (and processing of) *shared business events*. Event notifications 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. Yet, this broadcasting paradigm is fully compatible with current web service standards such as SOAP, WSDL and UDDI. The paper is structured as follows: Section 2 briefly overviews how the business event concept can be used at the level of *analysis*, *design* and *implementation* of web services. In Section 3, business events are used to enhance *web service description*, distinguishing between a (non-transaction) query interface and a (transactional) event notification interface. Section 4 discusses *web service composition and coordination*, again through participation in shared events. Conclusions are formulated in Section 5.

2 Event based web service development

The event based interaction mechanism as proposed in this paper directly reflects an underlying object-oriented analysis and design methodology: MERODE [3,4]. MERODE is complementary to UML [5], which can then be used as a *formalism* to capture the MERODE specifications. MERODE 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 "pure" method invocation to model interaction between domain object classes as in classical approaches to object-oriented analysis, e.g. [6]. 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, ORDERLINE and PRODUCT. The corresponding UML Class diagram is given in Fig. 1.

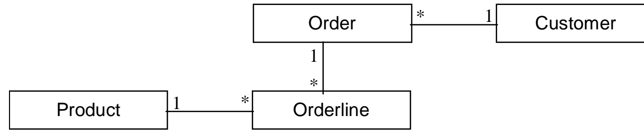

Fig. 1. Domain model for an order handling system

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 because the number of outstanding order lines has to be increased, and modifies the state of the CUSTOMER of the order because the total cost of outstanding orders has to be updated.

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		
bill	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

The event based domain model is combined with a *behavioral model*. Each enterprise object type has a method for each event type in which it may participate. Such method specifies *preconditions* put on the corresponding event type by the object type and implements an object's *state changes* (i.e. changes to attribute values) as the consequence of an event of the corresponding type. For example, when a customer orders a product, a new ORDERLINE is created, which involves the following business events: *create_order* and *create_orderline*. Preconditions may be based on *class invariants* (such as attribute constraints and uniqueness constraints) and on *event sequence constraints* that can be derived from a finite state machine associated with the object type. For example, Fig. 2 shows a finite state machine for the ORDER domain object. As long as it is not signed, an order stays in the state "existing". The *customer_sign* event moves the order into the state "registered". From then on the order has the status of a contract with the customer and it cannot be modified anymore: the events *modify_order*, create*_orderline, modify_orderline* and

end_orderline are no longer accepted for this order. The *ship_order* event signals that the order has been shipped to the customer, after which the order object reaches the "shipped" state. Finally, the *bill* event signals the billing of the order. In this way, the sequence constraints mimic the general business process(es). Full details of how to construct such an object-event table and validate it against the data model and the behavioral model are beyond the scope of this paper but can be found in [3,4].

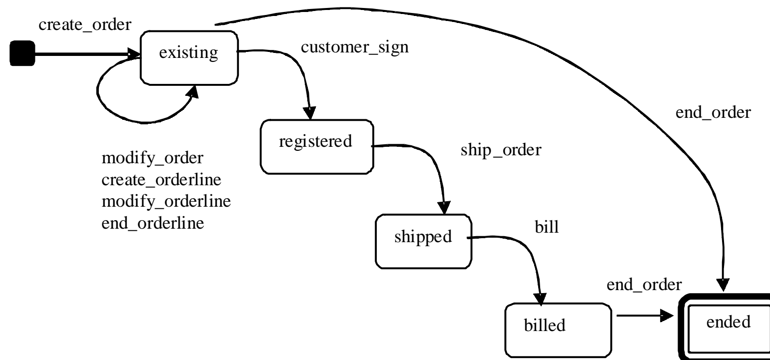

Fig. 2. State machine for an ORDER object

The entirety of all enterprise objects that together shape the business process(es) is called the *enterprise layer*. The eventual information system is realized as a layer on top of the enterprise layer, consisting of output and input services. *Output services* use attribute inspections to query the enterprise objects and deliver the information to the user. Upon occurrence of a business event in the real world, *input services* collect input data from the user and invoke the corresponding event to update the set of enterprise objects. This enterprise layer (and the associated input and output services) can be implemented in multiple ways: standalone or distributed, tightly coupled or loosely coupled. Currently, existing implementations are built around e.g. *stored procedures* [4] or an *EJB framework* [7].

As discussed in detail in [8], an implementation by means of web service technology can be achieved in three stages, in line with the Model Driven Architecture of the OMG [9]. First, the business rules are captured into a MERODE-based, technology neutral *business model*, which defines the enterprise objects and their interaction through participation in shared business events. The business model is then "enriched" into an *architectural model*, which groups the enterprise objects into distributed, loosely coupled components: the actual web services. Here, attribute inspection and event broadcasting are identified as distinct, complementary interaction types. In a third stage, the architectural model is translated into an actual technology-bound *implementation model*, based on current state-of-the-art technologies such as SOAP and WSDL.

A generic architectural model for a web services environment is depicted in Fig. 3. The enterprise layer is distributed across different services, with each service controlling/consisting of a subset of the enterprise objects. A web service is conceived as a layered structure: its local *enterprise layer* incorporates the actual business logic. This layer also contains *stub objects*, which locally represent external

web services. The layer above is called the *event layer* and consists of an *event dispatcher*, which also has a *transaction management* task. The highest layer is the *interface layer*. A web service can be accessed through two types of interfaces: a *query interface* and an *event notification interface*. Separate *input services* and *output services*, combined with a *user interface*, allow for human users to interact with the web service. For a thorough discussion of the proposed architecture, we refer to [8]. Throughout the remainder of this paper, the relevant components of the architecture will be clarified. The next section deals with both types of interfaces, as they will be the foundation to event based web service description.

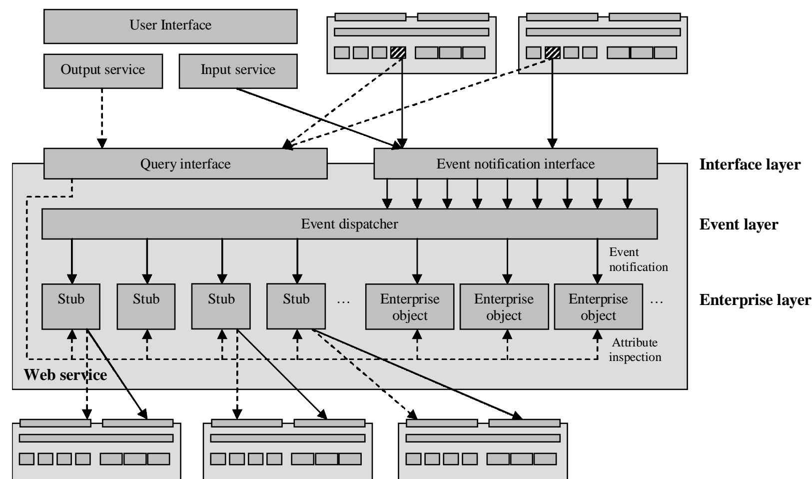

Fig. 3. A generic architectural model for a web services environment

3 Event based Web service description

3.1 Introduction

A key requirement for a successful web service is that potential users are able to find the right service and obtain the information necessary to interact with it. The way in which web services can be advertised and discovered strongly resembles the CORBA approach: CORBA objects are described by their *IDL interface*, which can be published in an *IDL repository* [10]. WSDL [11] and UDDI [12] can, to a certain extent, be considered as web service variants of respectively IDL and IDL repositories. However, whereas an intranet-based technology such as CORBA still pertains to a manageable number of services, transposing a similar approach to a world-spanning environment such as the Web may result in very poor searching performance. Specifying a service solely in terms of its *interface*, i.e. its input and output message types as is done in WSDL only offers a very incomplete picture, especially since the ultimate goal of many web services is to provoke changes in the *real world*, e.g. debiting a credit card in exchange for the delivery of a book at a certain address. This *business logic*, i.e. what the service "does" may be a much more

valuable search criterion. Although UDDI provides a categorization mechanism according to "real-world" criteria such as industry branch, product type and geographic location, this only accommodates for a rough, first-level filtering of available services. It is in no way destined at discovering a service based on fine-grained specifications of what is required from it. In this section, we discuss how the event based approach can make a web service's interface itself more descriptive by discerning between two interfaces: a *query interface* and an *event notification interface*. Further on, we indicate how the interface based description can be complemented with information about a service's *process logic*.

3.2 The query interface

The query interface allows reading the attribute values from the enterprise objects that make out a web service. Attribute inspection is a rather straightforward type of interaction and is based on a one-to-one process. The interaction does not cause a state change: the enterprise objects' attributes are never "written to" through a web service's query interface. Therefore, preconditions resulting from class invariants or sequence constraints are not applicable to attribute inspections[1], nor should such invocation be subject to a transaction context. In its simplest form, the query interface publishes a set of public *getAttribute()* methods at the service level, which are mapped transparently to *getAttribute()* methods on individual enterprise objects. Obviously, performance can be boosted if multiple correlated attribute inspections are bundled into a single SOAP message exchange. Therefore, in a more complex form, the query interface may allow invoking real *query methods* that inspect multiple enterprise objects in the web service and which may also calculate aggregate values, check for the existence of a certain object etc. In the latter case, a query method's input parameters behave as selection criteria that denote to which object(s) the attribute inspections apply.

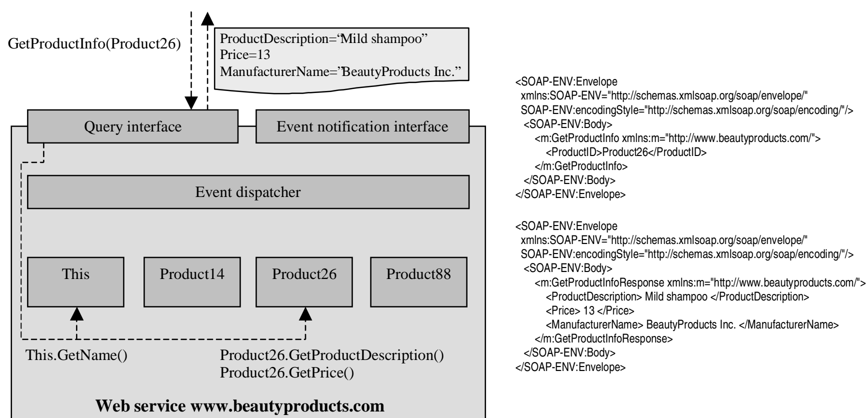

Fig. 4. Invocation on a web service's query interface

[1] The sole exception may be checks for the appropriate access privileges, which is beyond the scope of this paper.

Fig. 4 presents the simplified example (at instance level) of the query interface for a "BeautyProducts Inc." manufacturer web service. Invoking the *GetProductInfo()* method on the query interface results in the appropriate attributes being inspected on several enterprise objects, as determined by the method's input parameter "Product26". The requested values are communicated in a single return message. The example includes a (simplified) SOAP message that represents an invocation to this interface, along with a return message that contains the query result.

3.3 The event notification interface

The situation where a service A "writes" to a service B, i.e. causes attribute values in B's enterprise objects 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 enterprise objects that participate in the event. These combined updates must be considered as a single transaction. A service is never allowed to directly invoke *setAttribute()* methods on another service's enterprise objects. A service can only "write to" another service by inducing a *business event* on this service, which may affect the state of one or more enterprise objects embedded in the service, provided that all constraints are satisfied.

An event is induced by invoking the appropriate method on a web service's event notification interface. If a relevant event occurs in the real world, e.g. a customer issues a purchase order, a stock drops below a threshold value, an order is shipped, … this event is acknowledged by an input service, e.g. by a user pressing the "sign" button in a sales order form. The input service notifies the web service by an invocation on the web service's event notification interface. For each event type "understood" by the service, the event notification interface has a separate method. Attributes that describe the event (e.g. order quantity) are passed as input parameters to the method.

Upon invocation of such method, the event notification interface passes control to the *event dispatcher*. The latter implements a "local" OET and knows which event types are relevant to which (types of) local enterprise objects. The event is *broadcast* by the event dispatcher by simultaneously invoking the appropriate method on each enterprise object that participates in the event. The service's global reaction to the business event corresponds to the combined method executions in its individual enterprise objects. The corresponding transaction is only committed if none of the objects that take part in the event have generated an exception because of a precondition violation.

An example is, again at instance level, presented in Fig. 5. The web service for "BeautyProducts Inc." is notified of a *create_orderline* event for product #26, order #56 and with a quantity of 30. In *this* web service, only the enterprise object "Product26" is affected by the event. As will be depicted in Fig. 7, other enterprise objects in other web services can also be involved. The event is broadcast by the event dispatcher to Product26. The latter updates its stock level, provided that the constraint "order_quantity $\leq$ stock" is satisfied. Included in the example is a sample SOAP message that represents a corresponding event notification. Note that, in contrast to traditional method based interaction, no return values are sent other than possibly a status variable for transaction management purposes, i.e. to inform the

event dispatcher whether the event was accepted or whether it failed because a constraint was violated.

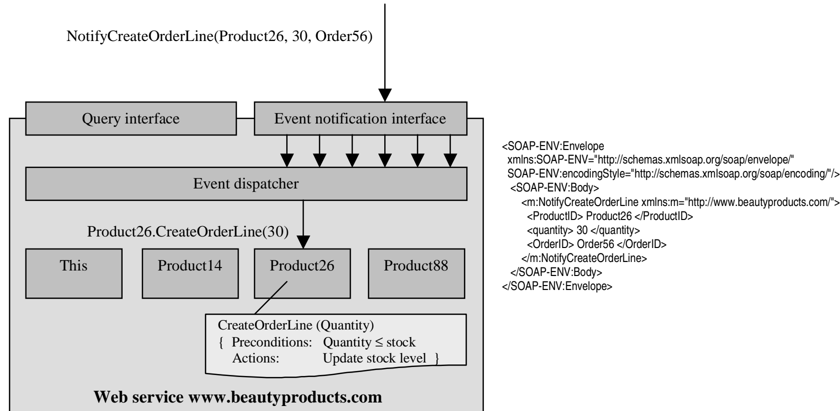

Fig. 5. Invocation on a web service's event notification interface

3.4 Discovery and invocation of web services

A "traditional" SOAP based method invocation on a web service may have two effects: it may cause a state change in the web service (as part of the *execution* of the method) and it may retrieve information from the service (by means of its *return value*). The *event* paradigm differs from this standard method invocation approach in that the cases of "*reading from*" a web service and "*writing to*" a web service are strictly separated. The latter is called the *Command-Query Separation* design principle, which, as discussed in general in [13], makes software much more transparent and makes classes easier to (re)use. [14] discusses its design and maintenance advantages specific to a web services context.

However, the distinction between a query interface and an event notification interface also allows for richer web service description. The query interface describes which information can be provided by the service. The interface not only *describes* the service, it can also be *invoked* by search engines to use the web service's current state as part of the selection criteria when searching for a web service, e.g. to search for an on-line library, which currently has the title ("XML for dummies") available.

In contrast, the event notification interface denotes to which types of real world events the service will respond, hence provides information about *what the service does*. The event paradigm has the advantage of reconciling a business concept with 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 services can *subscribe*. This business aspect of events does not reduce the description of an event notification interface to mere a software concept, but also links it to "real world events".

Both a web service's query interface and event notification interface are fully compatible with current web service standards. 'Traditional" web services or search engines will be confronted with a standard WSDL interface description. An example of a (partial) WSDL description of the BeautyProducts service's interface is depicted in Fig. 6.

```
<definitions name="BeautyProductsService"
 xmlns:s="http://www.w3.org/2001/XMLSChema/"
 xmlns:s0="http://www.beautyproducts.com/"
 targetNamespace="http://www.beautyproducts.com/">
  <types>...</types>
  <message name="GetProductInfoSoapIn">
     <part name="parameters" element ="s0:GetProductInfo"/>
  </message>
  <message name="GetProductInfoSoapOut">
     <part name="parameters" element ="s0:GetProductInfoResponse"/>
  </message>
  <message name="NotifyCreateOrderLineSoapIn">
     <part name="parameters" element ="s0:NotifyCreateOrderLine"/>
  </message>
  <portType name="ManufacturerPortType">
     <operation name="GetProductInfo">
       <input message="s0:GetProductInfoSoapIn"/>
       <output message="s0:GetProductInfoSoapOut" />
     </operation>
     <operation name="NotifyCreateOrderLine">
       <input message="s0:NotifyCreateOrderlineSoapIn"/>
     </operation>
  </portType>
  <binding>...</binding>
  <service>...</service>
</definitions>
```

Fig. 6. Example of a partial WSDL description for the BeautyProducts web service

Once the appropriate web service has been selected, one needs to be able to retrieve information about how a request to the service is to be conducted. Traditionally, this will include details of the input parameters that are to be provided and the output parameters that can be expected. The latter will suffice for very simple services, e.g. a currency conversion service, but it may prove to be limiting in the case where multiple types of inputs and intermediate conditions may result in a diverse range of possible output types, e.g. an online trip booking service. In such case, an explicit description of the service's *logic* may be required. Also, "real world" properties, which cannot be considered input or output parameters but which may definitely affect the outcome of a transaction, should be taken into consideration. As to these, the event based approach can again be utterly useful, simply because the event notion bridges the gap to real world concepts.

Taking this one step further, one could imagine the web service'*s state machine* being published as metadata to the service, as applied in DAML-S [15]. An event may induce a state transition to one or more of the service's objects (and, as discussed further on in this paper, to one or more services). The published state machine would be a unified version of the respective state machines of the internal enterprise objects, retaining only these states and transitions that are relevant to the outside world. In this way, the calling application would know in advance which preconditions are required for a certain event to be accepted and which results (i.e. postconditions) can

be assured. In contrast, an invocation to the query interface will never induce a state transition.

Finally, such event based description lends itself well to be enhanced with semantic web concepts. Ontology languages such as OWL [16] can be used to describe the "meaning" of a certain business event type, parameters, states etc. at a semantic level, as advocated in [17].

4 Event based web service coordination, composition and choreography

4.1 A composition model for static B2Bi

As already stated, the web service concept hasn't fulfilled its full potential (yet) at the level of complex, composite services. Standards for web service *transactions*, *composition* and *choreography* are still under development. Existing attempts such as BTP [18], WSCL [19], BPEL4WS [20] or BPML [21] are based on quite divergent assumptions. This paper does not try to introduce yet another standardization proposal. Rather, it discusses how the event broadcasting concept may provide a composition model that facilitates the development of a standard.

In our opinion, an important factor in the troublesome process of web service coordination and choreography is the fact that SOAP, as it is traditionally used, is essentially a *one-to-one mechanism*. The latter may be adequate 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. With an event based paradigm, transactional interaction, i.e. a "write" operation is essentially a *broadcasting* mechanism instead of a one-to-one method invocation: event broadcasting is *implemented* by means of SOAP method invocations, but the methods *are executed in parallel and in a coordinated way on all enterprise objects that participate in the event*. In this way, a single business event results in multiple simultaneous updates in multiple enterprise objects. Previous sections discussed this issue at the level of a single service. This section denotes how events can mould coordinated interactions *between* services, which are much easier to model than the myriad of one-to-one message exchanges that could make out a single business transaction in a pure method based approach.

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 enterprise objects that participate in the event, based on a "local" object event table. As to remote objects to which the event may be relevant, the proposed architecture caters 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 services that are subscribed to the corresponding event type. For that purpose, a web service's architectural model is enriched with *stub objects* that locally "represent" an individual remote service and that contain a reference to it, e.g. its URL. The general purpose of these stub objects is to make the distribution aspect of the enterprise layer transparent to a web service's local enterprise objects and to its event dispatcher. A first function of a stub object is to "mirror" attributes that belong to the enterprise objects of the

remote service it represents (see [8] for more details). However, more importantly, it also *propagates* event notifications to the remote service it represents. As illustrated in Fig. 7, the resulting event based interaction mechanism takes place in four stages: if an event occurs in a given web service, as initiated by an invocation to its event notification interface (1), this event is broadcast to all appropriate *local* objects, i.e. enterprise objects and stub objects (2). Each stub object propagates the event by invoking the appropriate method on the event notification interface of the external service it represents (3). Such remote service's event dispatcher then in its turn broadcasts the event to its own local objects (4). Some of these may also be stub objects that further propagate the event etc. The appropriate enterprise objects each execute a corresponding method, in which preconditions are checked and/or updates are executed (5).

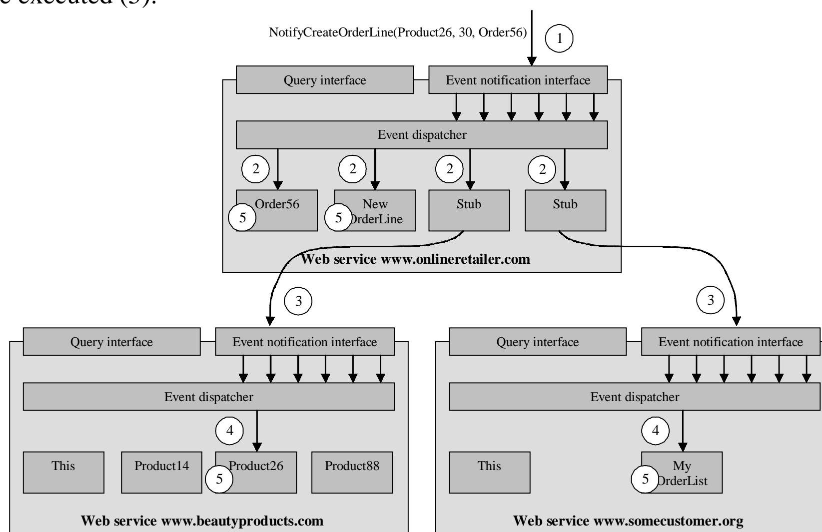

Fig. 7. Example of event propagation by means of stub objects

Stub objects play an important role in distributed transaction management and web service choreography. Indeed, within an individual service, the event dispatcher enforces the "atomicity" of an event: an event only succeeds if no preconditions are violated in any of the enterprise objects that participate in it. The event dispatcher commits the transaction if no "reject" messages were returned by any local enterprise object. Local stub objects may also reject an event: a stub object does not impose preconditions itself, but receives an accept/reject return value upon invocation to the event notification interface of the remote web service it represents. If the (partial) transaction fails in the remote service (because preconditions were not met in one or more of the latter's own enterprise objects), the stub receives a "reject" return value. As a consequence, the stub rejects the event locally and in its turn returns a reject value to the local event dispatcher. In this way, the concept of event broadcasting and propagation results in a distributed transaction mechanism. Whereas the underlying SOAP invocations only represent one-to-one interactions, events can be seen as coordinated actions that affect multiple services. They are initially induced on a single service, but are propagated to all services subscribed to the event (event

subscription is discussed in section 4.2) and finally result in parallel method invocations on all (local and remote) enterprise objects that participate in the event. The action is only committed if no constraint are violated in any of the participating services/enterprise objects. In this way, events shape a coordination mechanism, derived from the business model, over the individual SOAP invocations.

In this way, event propagation can be used both for interaction between peer services (by the stubs propagating event notifications) and for a complex service to coordinate the behavior of its components (by the local event dispatcher notifying local objects). This approach can be easily generalized into an n-level system: each local object and each remote service may in their turn be complex objects, which react to the event by further propagating it to their constituent objects or (through their own stubs) to yet other remote services etc. 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. The choreography is not stored in a central document: the appropriate consecution(s) of events is/are determined by the preconditions imposed by the individual enterprise objects. These preconditions emanate from the state machines of the enterprise objects, hence reflecting the actual business processes.

4.2 A composition model for dynamic B2Bi

The previous section actually dealt with a model for *static* B2Bi. In such situation, all interacting partners "know" one another in advance: together they form an extended enterprise. In such situation, dynamic event subscription is not necessary; stub objects are created at the moment when the interacting web services are deployed.

This approach can be extended into a composition model for *dynamic* B2Bi, where partners have to dynamically *find* one another, after which they participate in short lived, ad hoc partnerships. In that case, stub objects will be created at runtime, when a certain remote service is selected for interaction. Just like the "real" enterprise object types, stub object types are represented in the object-event table. The latter denotes which event types have a "create" effect on a stub object type. If such an event is induced, a new stub object instance is created (at least if all preconditions are satisfied) and initialized with the URL of the external service it represents. This URL is to be provided as one of the event's parameters. From then on, the stub object is responsible for the interaction between the two services. In this way, dynamic subscription of one service to another, i.e. the initiative to start interacting, is modeled as part of the business processes and embedded in the lifecycle of the enterprise objects. In a similar way, events with an "end" effect on a stub object terminate the interaction between two services.

In this respect, one last issue is how the "decision" of one service to search for and subscribe to another service is to be made. This paper does not address the search (and possible matchmaking) mechanism itself, but its contribution lies in the ability to automatically formulate web service search criteria through the concept of *failed events*. Indeed, an event can be seen as a transaction. Web transactions typically have transaction management mechanisms that are less strict than the traditional "ACID" properties from the database world. Especially, "long -lived" transactions (e.g. the online booking of a trip) 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. In the context of events, a failed transaction can be translated as a precondition not being satisfied in one of the enterprise objects (either local or remote). Instead of rolling back the entire event, a measure of *goal seeking* intelligence can be added to the event dispatcher so as to come up with a corrective action for the failed precondition, instead of aborting the entire transaction. Two alternatives can be discerned, which can be enhanced with semantic web based mediation capabilities as discussed in [22]. A first possibility is that the event dispatcher of the web service where the event was initially induced tries to induce (an) additional, "subordinate" event(s) on the service where the event was refused. The subordinate event(s) should result in state changes that resolve the original precondition violation. In this respect, the precondition can be seen as the *goal* that is to be achieved to be able to have the original event succeed. The subordinate events can be selected based on whether their *postconditions* assist in achieving this goal. For example, a *purchase* event in an on-line shop could be rejected because it is "membership required". The calling service could then try to induce a *new_membership* event, so as to satisfy the precondition for the original purchase event. A second option is to simply "replace" the service where the precondition was violated with a similar service that imposes less strict preconditions. In that case, the new service will be selected based on the events it understands *and* on the preconditions it imposes on them. For example, the calling service could look for another shop, which does not require membership. Again, search criteria not only entail interface formats, but also specifications about the effect(s) on the real world.

5 Conclusions

A rigorous analysis and design phase is often overlooked with respect to web service development [23]. This paper proposed a business modeling paradigm where an information system's behavior is closely related to real world *business events*. The relation between static and dynamic specification is made by means of a very elegant and concise modeling technique: the object event table. The event based specification can be *implemented* without giving up on the current de facto standard web services stack based on SOAP, WSDL and UDDI.

Moreover, the clear distinction between query interface and event notification interface results in improved web service description capabilities. The description can be further enhanced by an explicitation of the *business logic* by means of state machines, with business events inducing the state transitions.

Events also facilitate the coordination of interacting web services. Discerning attribute inspections from event notifications allows for focusing on only the latter with respect to transaction management. Also, the fact that event propagation is a broadcasting mechanism yields a composition model that is much simpler than streamlining the myriad of one-to-one message exchanges in a purely RPC based approach.

Not unlike [24], statecharts are used to capture the business processes, rather than flowcharts as applied e.g. in [25]. Particular to our approach is that *business events* are a core modeling component, triggering and coordinating state transitions in multiple enterprise objects and services. The "choreography" is distributed as

preconditions imposed by the respective enterprise objects and services that participate in an event; at each level the behavior of a composite system is the union of the individual objects' behavior. Dynamic B2Bi is facilitated by incorporating a goal seeking and subscription mechanism, again based on business events and preconditions resulting from the state machines that reflect the underlying business processes.

References

1. Leclerc, A.: Distributed Enterprise Architecture, Integrating the Enterprise, Vol. III, no. 5 (2000)
2. Seely, S., Sharkey, K.: SOAP: Cross Platform Web Services Development Using XML, Prentice Hall PTR, Upper Saddle River, NJ (2001)
3. 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)
4. Snoeck, M., Dedene, G., Verhelst M, Depuydt A. M.: Object-oriented Enterprise Modeling with MERODE, Leuven University Press, Leuven (1999)
5. OMG, Unified Modelling Language, http://www.omg.org/uml/
6. 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)
7. Lemahieu, W., Snoeck, M., Michiels C.: An Enterprise Layer Based Approach to Application Service Integration, Business Process Management Journal, (forthcoming)
8. Lemahieu, W., Snoeck, M., Michiels, C., Goethals, F., Dedene, G., Vandenbulcke, J.: A Model Driven, Layered Architecture for Web Service Development, K.U.Leuven – F.E.T.E.W. internal research paper, currently under review at IEEE Computer (2003)
9. OMG, Model Driven Architecture, http://www.omg.org/mda/
10. Siegel, J.: CORBA – fundamentals and programming, John Wiley & Sons, New York, NY (1996)
11. Web Services Description Language (WSDL) 1.0. specification, http://msdn.microsoft.com/xml/general/wsdl.asp (2001)
12. UDDI Technical White Paper, Ariba, IBM Corporation and Microsoft Corporation, (2000)
13. Meyer, B.: Object-oriented software construction, second edition, Prentice Hall PTR, (1997)
14. Lemahieu, W., Snoeck, M., Michiels, C., Goethals, F.: An Event Based Approach to Web Service Design and Interaction, accepted for APWeb'03, LNCS 2642 (2003)
15. DAML-S official website, http://www.daml.org/services/daml-s/2001/05/ (2001)
16. Dean, M., Schreiber G., Van Harmelen, F. et al.: OWL Web Ontology Language 1.0 Reference, W3C Working Draft (2003)
17. 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)
18. Potts, M., Cox, B., Pope, B.: Business Transaction Protocol Primer, OASIS Committee Supporting Document (2002)
19. Banerji, A. et al.: Web Services Conversation Language (WSCL) 1.0, W3C Note (2002)
20. Weerawana, S., Curbera, F.: Business Process with BPEL4WS, IBM white paper (2002)
21. Arkin, A.: Business process Modeling Language, BPMI draft specification (2002)
22. Fensel, D., Bussler, C., Ding, Y., Omelayenko, B.: The Web Service Modeling Framework WSMF, Electronic Commerce Research and Applications 1(2), (2002)
23. Frankel, D., Parodi, J.: Using Model-Driven Architecture to Develop Web Services, IONA Technologies white paper (2002)

24. Benatallah, B., Dumas, M., Sheng, Q., Ngu, A.: Declarative composition and peer-to-peer provisioning of dynamic web services, ICDE 2002 (2002)
25. Casati, F., Jin, L., Ilnicki, S., Shan, M.C., An open, Flexible and Configurable System for Service Composition, HPL Technical report HPL-2000-41 (2000)