Information and Software Technology 52 (2010) 1023–1043

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

WSDL and BPEL extensions for Event Driven Architecture

Matjaz B. Juric

University of Maribor, FERI, Laboratory of Communications Technologies, Smetanova 17, SI-2000 Maribor, Slovenia

ARTICLE INFO

Article history:
Received 10 July 2009
Received in revised form 10 March 2010
Accepted 27 April 2010
Available online 20 May 2010

Keywords:
Event
EDA (Event Driven Architecture)
SOA (Service Oriented Architecture)
WSPA (Web Services Platform Architecture)
BPEL (Business Process Execution Language)
WSDL (Web Service Description Language)

ABSTRACT

Context: Service Oriented Architecture (SOA) and Event Driven Architecture (EDA) are two acknowledged architectures for the development of business applications and information systems, which have evolved separately over the years.
Objective: This paper proposes a solution for extending the SOA/Web Services Platform Architecture (WSPA) with support for business events and EDA concepts. Our solution enables services to act as event producers and event consumers. It also enables event-driven service orchestrations in business processes.
Method: Based on a comparison of SOA and EDA, we have identified and designed the required extensions to enable support for events and event-driven process orchestration in WSPA.
Results: We propose specific extensions to WSDL and BPEL, and a flexible XML representation of the event payload data. We introduce event sinks, sources, and triggers to WSDL. We extend BPEL with new activities to trigger and catch events, and extend fault and event handlers, variables, and correlation properties to accommodate events.
Conclusion: As a proof-of-concept, we have developed a prototype implementation and assessed the extensions on three pilot projects. We have shown that our proposed extensions work on real projects and that combining event-driven and service-oriented semantics makes sense in many business applications and can considerably reduce the development effort.

1. Introduction

The objective of this article is to propose a solution for support of business events and EDA (Event Driven Architecture) concepts in Service Oriented Architecture (SOA). To achieve this we propose specific extensions to WSDL (Web Service Description Language) and BPEL (Business Process Execution Language).

Services and processes in SOA, implemented using Web Services Platform Architecture (WSPA), use an operation invocation model and support different message exchange patterns, among them the request-response and one-way operation invocations. They support synchronous and asynchronous operation invocations. Services are accessed via remote interfaces. Remote interfaces are central to the design of SOA/WSPA applications. The interaction with services is in most cases a two-party interaction where the client (service consumer) invokes operations on the service (service provider) using the service interface. EDA architectural style is based on the notion of business events. The communication between distributed components in EDA is achieved through posting and receiving events and is a multi-party interaction. Events are central to the design of EDA applications. Events are intended to be usable through different components and applications.

SOA and EDA both serve the same purpose, development of distributed modular business applications. The major difference between them is the approach to the composition (orchestration or choreography) of business components into larger logical units of works (such as processes and workflows), where SOA is based on the operation invocation style and EDA on the event notification style. Both invocation and notification styles have their advantages and disadvantages.

SOA and EDA have evolved as separate architectures. Today both architectures are acknowledged, but their synergy is not. In this paper we will argue that synergy between the SOA and EDA is meaningful. There are numerous benefits of having an architecture that supports coexistence between operations and events, and composition of services based on operation invocation and event triggering. Therefore, in this article we introduce specific extensions to WSDL 2.0, 1.1, and BPEL 2.0 that provide support for events in WSPA and allow composition of services based on event triggering. With the proposed extensions, events become first-class citizens and make services and processes appropriate for complex event processing (CEP). The major benefit of introducing events to SOA/WSPA is the ability to further reduce the level of coupling between services and processes. Proposed extensions have positive effects on key service design principles [7], including improved loose coupling due to reduced dependencies between services, better service abstraction due to improved hiding of the underlying

E-mail address: matjaz.juric@uni-mb.si

0950-5849/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2010.04.005

service details, improved service reuse due to improved positioning of services as enterprise resources with agnostic functional contexts, and more flexible service composability that is based on events rather than on operation invocations.

The proposed extensions to WSDL introduce events, event sources, and event sinks. They allow services to listen to events and they allow services to be event producers. For event listeners, filtering and event triggering based on operation invocation is supported. The proposed extensions to BPEL introduce the ability to orchestrate event-capable services. Business processes that use extended BPEL can react on events within the orchestrations and they can trigger events. To achieve this, new activities are introduced and existing activities are extended.

This article is organized in nine sections and two appendixes. In Section 2, we compare SOA and EDA and identify key differences and their implications. In Section 3, we explain the motivation for combining SOA and EDA and present the high-level idea. In Section 4, we give a brief overview of WSDL and BPEL, which we will extend with events. In Section 5, we describe the proposed WSDL extensions for events, including event declarations, event sources, event sinks, event triggers, and bindings. In Section 6, we describe the proposed BPEL extensions for events, including extensions to variables, extensions for triggering and catching events, event handlers, fault handler extensions, and correlation and property alias extensions. In Section 7, we present the proof-of-concept, where we describe the implementation of the proposed extensions for events. In Section 8, we present related work and discuss the results. In Section 9, we give conclusions. Appendix A defines syntax specification for WSDL extensions for events. Appendix B defines syntax specification for BPEL extensions for events.

2. Comparison of SOA and EDA

The componentization and the related modularity of software have become increasingly important over the last years. Particularly in the development of information system and business applications, componentization has been pursued for many years. Business components – independent, autonomic, and reusable modules that represent a unit of work and have a meaning in a business context – have become the predominant approach to architecting modular, distributed information systems [1]. Although the concept of business component has become commonly accepted there are significant differences in the approach, how business components interact with each other. Two major approaches have emerged [65–67]. The first one is the family of approaches, which are based on the notion of operations and their invocation. The second one is the family of approaches, which are based on the notion of events, and on triggering and consuming the events.

The first family has its roots in remote procedure call (RPC) techniques [2,68,69], although the notion of business components has emerged later with DCE (Distributed Computing Environment) [3] and CORBA (Common Object Request Broker Architecture) [4]. Their most well known successors have been COM+ (Component Object Model), RMI (Remote Method Invocation), and EJB (Enterprise Java Beans) [5]. Out of them, web services, WSPA and SOA have emerged.

The common denominator among all these architectures has been the interaction pattern between components, which has been based on operation invocations. The interaction involves two parties, the business component and the client. The client requests (calls or invokes) an operation on a specific component and the component fulfills it. If a request-response interaction pattern is used, the component will return a result to the client. If a one-way interaction pattern is used, the component will not return a result or will return it using a separate operation invocation, often called callback. Particularly the request-response pattern has often lead to conversational interaction style between components and has resulted in relatively tight-coupled systems with low autonomy of individual components. This has limited the flexibility of systems and hindered reuse.

SOA is the latest representative of this family [6]. It is important to distinguish between the SOA as a concept, and the implementation. SOA concepts can be realized using different technologies. The current de facto realization technique for SOA is WSPA [63], which uses the web services technologies and is better aligned to SOA concepts as the older architectures. SOA has introduced services as main building blocks. To facilitate operation invocations between services, SOA introduced the ESB (Enterprise Service Bus) [71]. In SOA/WSPA a lot of effort has been directed towards achieving compliance with service design principles, including loose coupling, abstraction, reuse, and service composability [7] in order to show a visible improvement over older architectures. The notion of loose-coupling precludes any knowledge or assumptions about the implementation of the services, the formats and protocols used to interoperate between them and the specific platforms that the service consumer or the service provider run on [63]. To improve loose coupling, SOA/WSPA has tried to reduce dependencies between services by introducing document-style interaction instead of conversational interaction style used by older architectures that lead to tighter coupling. Service abstraction has been improved by decoupling the service interface from the underlying service details.

SOA/WSPA has been only partially successful in fulfilling service design principles, particularly due to the limitations of the operation invocation interaction pattern, which is based on the invocation of a specific operation on a specific component (service) interface. In WSPA service interaction patterns are defined in WSDL and BPEL. WSDL 1.1 has introduced four operation types: request-response, one-way, solicit-response, and notification. Only the first two operation types are used in practice [64]. BPEL can be used to define more complex interaction patterns [70]. BPEL 2.0 is based on WSDL 1.1. WSDL 2.0 has introduced a generic mechanism to define Message Exchange Patterns (MEPs). However, the above-mentioned objectives have not been fully achieved. This makes it more difficult to integrate components [8] and to involve other interested parties into the interaction between the client and the service. Fig. 1 shows a graphical representation of an operation invocation.

The second family, the EDA approach, is based on the production, detection, consumption, and reaction to business events. Business events represent a significant change in business state [9]. Events require a bus, which distributes the events from its source component (event producer) to the sink components (event consumers). This can be message-oriented middleware, such as JMS (Java Message Service), MQ Series, and MSMQ [10]. Or it can be a proprietary event processing infrastructure, such as Oracle CEP [11]. It could also be the ESB (Enterprise Service Bus) [72], which would have to be extended to support events. Event-driven approach is based on strict separation between the event producer and event consumer [12]. A single event producer can have multiple event consumers. When the event producer triggers an event, the event is delivered to all event consumers that have registered (subscribed) for such event. This is a major difference compared to operation invocation, where the client invokes an operation on a single specific service interface. The fact that any number of event consumers can subscribe to the event leads to a very high degree of loose coupling. The communication between event producers and event consumers is usually one-way only and asynchronous and follows the notification-subscription model. Therefore, the default interaction style in EDA is document-style. This results in greater level of component autonomy in EDA and

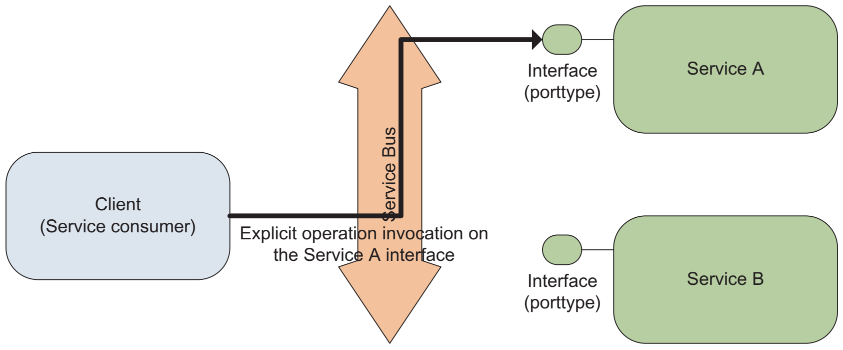
Service Bus
Interface (porttype)
Service A
Client (Service consumer)
Explicit operation invocation on the Service A interface
Interface (porttype)
Service B

Fig. 1. Client invokes an operation on a service.

potentially in higher composability [13]. On the other hand, the event-based communication has hindered the ability to design truly reusable business components, which negatively influenced reuse [1]. The event driven communication has also resulted in increased complexity of interactions, which are less transparent and more difficult to overview and maintain. Fig. 2 shows a graphical representation of an event producer and several event consumers.

3. Motivation for combining SOA and EDA concepts

Table 1 shows a brief comparison of the most important concepts between SOA (using WSPA) and EDA. Both SOA and EDA styles have developed separately over the years. Combining SOA and EDA concepts would be reasonable, as it would provide the designers and the developers the best of both worlds.

To combine SOA and EDA concepts, we propose to extend WSPA services with the notion of events. Events in the extended WSPA can use the XML representation of the associated data, which makes event structure flexible, maintainable, and easy to understand. Services in WSPA, which expose operations through interfaces (port types), can be extended to act as event producers and event consumers (Fig. 3).

ESB can be extended to take over the role of event dispatcher. To orchestrate services the Business Process Execution Language (BPEL) is used [14]. BPEL orchestrations are based on operation invocations on service interfaces. BPEL can be extended to support events and enable orchestrations (compositions) of services based on events. This way the developers can use the best of both worlds and combine the operation invocation style with the event driven style. Fig. 4 shows an example scenario, where a BPEL process orchestrates Service A and Service B. BPEL triggers event A through event source. Services A and B are subscribed to event A and receive the event through event sinks. BPEL also invokes a request-response operation on the Service A interface (portType). BPEL process can also expose a service interface (for example for callbacks from services). As a response to event A, Service A triggers event B. Service B is subscribed to event B. So is the BPEL process, which can also act as an event consumer.

Table 1
Comparison between SOA and EDA.

	SOA (using WSPA)	EDA
Interaction model	Operation invocations	Event triggering
Dependencies between participants	Interface	Event
Relationship between participants	Client knows the service interface and the operation to invoke	Event producer knows nothing about the event consumers
Availability of participants	Service has to be available when the client invokes it	Event consumers do not need to be available at the time event is triggered
Interaction patterns	Defined by Message Exchange Patterns (MEPs)[a]	One-way
Interaction style	Conversational and document	Document
Level of coupling	Tight to loose	Loose to very loose
Bindings	The client binds to the service interface	Event producer does not bind to event consumer
Extension approach	Close-ended: clients have to be modified to call new service operations or new service interfaces	Open-ended: event producers do not need to be modified to include new event consumers
Contract between participants	Service interface and operation signature	Event name and event payload structure

[a] WSDL 1.1: request-response, one-way, solicit-response, notification; WSDL 2.0 provides a generic mechanism to define MEPs, eight are predefined: in-only, robust in-only, in-out, in-optional-out, out-only, robust-out-only, out-in, out-optional-in. BPEL can be used to define more complex interaction patterns, however please note that BPEL 2.0 is based on WSDL 1.1.

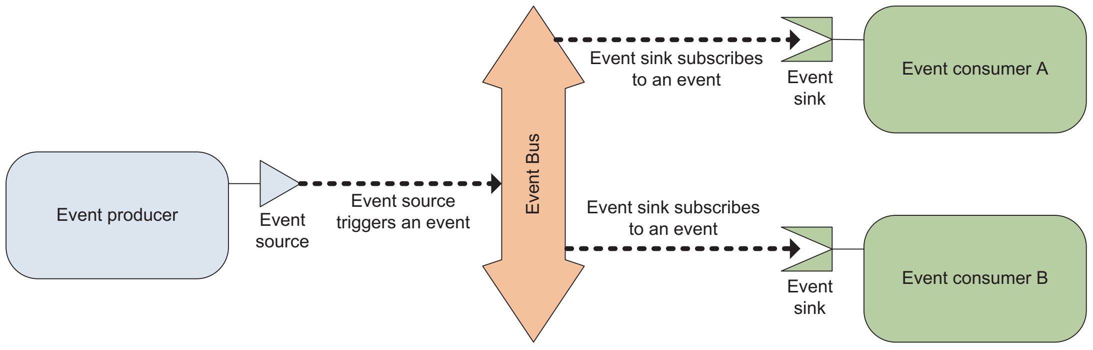
Event sink subscribes to an event
Event sink
Event consumer A
Event Bus
Event producer
Event source
Event source triggers an event
Event sink subscribes to an event
Event sink
Event consumer B

Fig. 2. Event producer triggers an event.

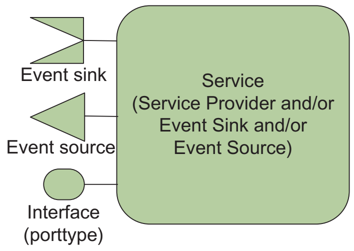
Event sink
Event source
Interface (porttype)
Service (Service Provider and/or Event Sink and/or Event Source)

Fig. 3. Extended service acting as service provider, event sink and event source.

To realize such described approach, we propose to extend the WSDL and the BPEL, the two most important WSPA languages, with event support. We also propose how to define the event payload using XML. We have designed the extensions so that the events in WSPA become first-class citizens, not just an add-on. The developers can freely combine event-driven approach and operation-invocation approach. We propose a way to incorporate event-driven principles into service description using WSDL, so that services can become event producers and consumers. We also propose extensions to BPEL to allow service orchestrations to accommodate events and event-driven semantics. We propose extensions to BPEL, which go way beyond the existing support for events in BPEL. BPEL currently provides event handlers, which are used to enable a BPEL process to react on operation invocation events. They are triggered by the incoming operation invocation messages. Our approach introduces business events. Handling such events from BPEL differs from reacting on operation invocation message events, as we will describe later in this article. Before we present the extensions let us have a brief look at WSDL and BPEL.

4. Brief overview of WSDL and BPEL

WSDL is a W3C (World Wide Web Consortium) recommendation and has become the de facto standard for defining web service interfaces [15,16]. WSDL is used to describe the interfaces of all services irrespective of the underlying technology. In June 2007, W3C has published the WSDL 2.0 recommendation [16]. At the time of writing, not many vendors have adopted version 2.0. It is not clear whether version 2.0 will gain the necessary support [17,18]. Therefore, we have designed the extensions for both WSDL versions 2.0 and 1.1.

WSDL 1.1 [15] describes the interface of a service within the <*portType*> section, where the operations are listed. WSDL supports one-way and two-way operations. Each operation is defined by *input*, *output* and optional *fault* messages. The input message represents the payload an operation receives in order to perform the processing, while the output message represents the resulting payload. The fault message is used to signal faults. Output and fault messages can only be used in request-response operations. Each message consists of several parts. Message parts are defined within the <*message*> section. Each message part is defined by the corresponding schema as a simple XML type, a complex type or an element. The schema can be found in the WSDL document under the <*types*> section. Schema can be embedded in the WSDL document, or can be imported from a standalone schema (XSD) document. The <*binding*> defines the mapping of the interface/payload to the corresponding protocol, such as SOAP, or any other supported protocol. Within the binding part we also define the style (document or RPC) and the payload representation (literal or encoded). The <*service*> defines the service endpoint location.

WSDL 2.0 [16] has simplified the interface description. The interface is defined within the <*interface*> section. WSDL 2.0 has dropped the concept of input and output messages. Rather the operations directly specify XML elements for input and output. For faults, fault messages can be specified within the <*interface*> section. This allows that the same fault message is used in different operations, which simplifies fault handling. WSDL 2.0 has introduced the message exchange patterns, which specify the sequence in which the associated messages are to be transmitted between the service and the client. WSDL 2.0 also allows interface inheritance. The <*binding*> and <*service*> parts have not changed significantly apart from the new syntax. WSDL 2.0 has retained support for literal style of web services only. Neither WSDL 2.0 nor 1.1 provides support for events.

BPEL is an OASIS standard [14] and has become the de facto standard for service orchestration. BPEL is supported by the majority of SOA/WSPA platforms and development tools [19]. It provides support for executable and abstract business processes. The

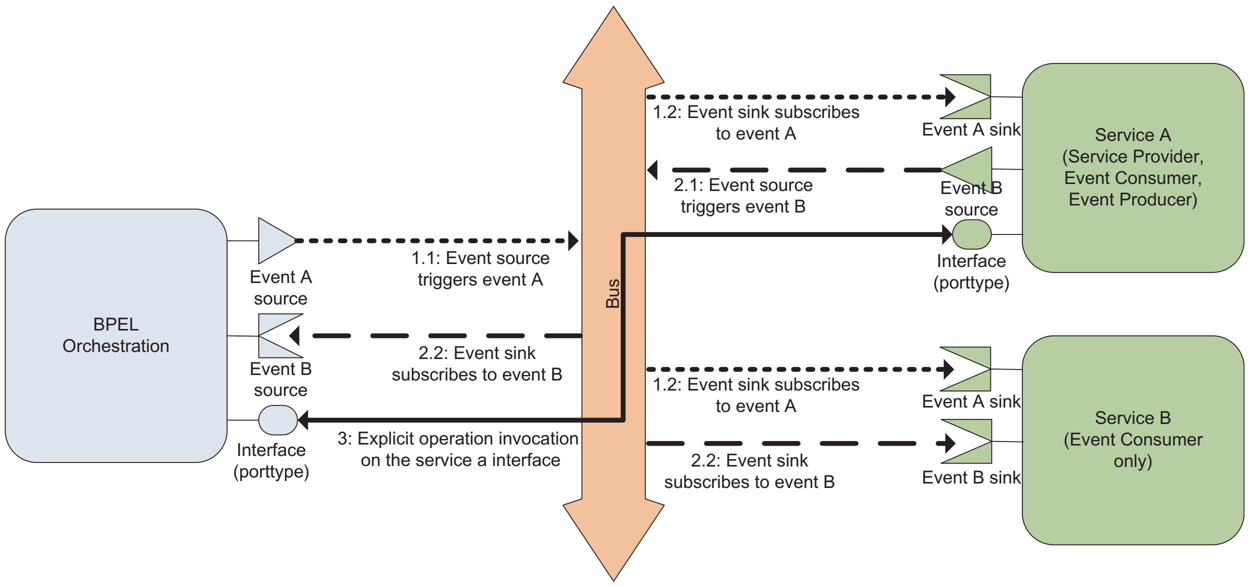
BPEL Orchestration
Event A source
Event B source
Interface (porttype)
1.1: Event source triggers event A
2.2: Event sink subscribes to event B
3: Explicit operation invocation on the service a interface
Bus
1.2: Event sink subscribes to event A
2.1: Event source triggers event B
Event A sink
Event B source
Interface (porttype)
Service A (Service Provider, Event Consumer, Event Producer)
1.2: Event sink subscribes to event A
2.2: Event sink subscribes to event B
Event A sink
Event B sink
Service B (Event Consumer only)

Fig. 4. Example scenario of orchestrating services using a combination of events and operation invocations.

current version is 2.0, which has been approved by OASIS in April 2007. BPEL 2.0 is based on WSDL 1.1. BPEL 2.0 is an evolution of the previous version 1.1 and introduces several improvements, such as improved variable manipulation, enriched fault handling, improved correlation, local partner links, dynamic parallel flows, improved loop handling, and extension mechanism, which allows adding extensions to the BPEL language in a standardized way [20]. We use the BPEL extension mechanism to add support for events.

5. Proposed extensions to WSDL

WSDL service description is operation-centric. Our objective has been to extend WSDL with the native support for events. This way a service (web service) would be able to produce events and/or receive events. A service could thus be an event source or an event sink. At the same time, the service should also retain the ability to expose operations. Therefore, we have decided to add new constructs to the WSDL. We use the XML namespace extension mechanism, which allows us to extend WSDL while assuring backward compatibility with tools that are unaware of the extensions. To achieve this we have introduced a namespace URI http://www.uni-mb.si/wsdl/eventExtensions, for which we use the alias *wsdlx*.

To enable support for a service to act as an event source or event sink, we have added the corresponding <*wsdlx:eventSource*> and <*wsdlx:eventSink*> constructs to the service interface (portType). The <*wsdlx:eventSource*> declares that a service is producer of events. The <*wsdlx:eventSource*> declares that a service is consumer of events. To trigger events we have added a <*wsdlx:eventTrigger*> construct.

5.1. Event declaration

Each event has three sets of event-specific data: headers, bodies and faults. Event headers are meant to carry system-related data regarding an event. Each event must have an obligatory header (discussed below). Event bodies are meant to carry business or application related data regarding the event. Faults carry data about exceptional states.

Event headers, bodies and faults are defined by the corresponding XML Schemas. We followed the principle to make the event structure as simple and as flexible as possible. Therefore, the only obligatory part of the event is the predefined header. It specifies the event name (required), unique ID (required), version (optional), creation date and time (required), validity from/until/duration (optional), and service name that triggered the event (required). Listing 1 shows the XML Schema for the obligatory event header.

In addition to the predefined header, an event can have zero to more custom defined headers. An event can also have zero to more custom defined bodies and faults. This way the definition of the event is flexible and allows extensions by adding additional headers, bodies and faults without modifying existing. This is useful for existing event sources and sinks that do not need to be modified if the event structure is extended, which assures backward compatibility.

To declare events, we have introduced the <*wsdlx:events*> construct. Within it we declare individual events using <*wsdlx:event*>, or we import external event declarations using <*wsdlx:import*>. The event declarations are located after the <*types*> section and before the <*message*> or <*interface*> section in the WSDL document.

For the <*wsdlx:event*> we have to specify the event *name*. Optionally we can specify the event unique ID (*uid*) and the *version*.

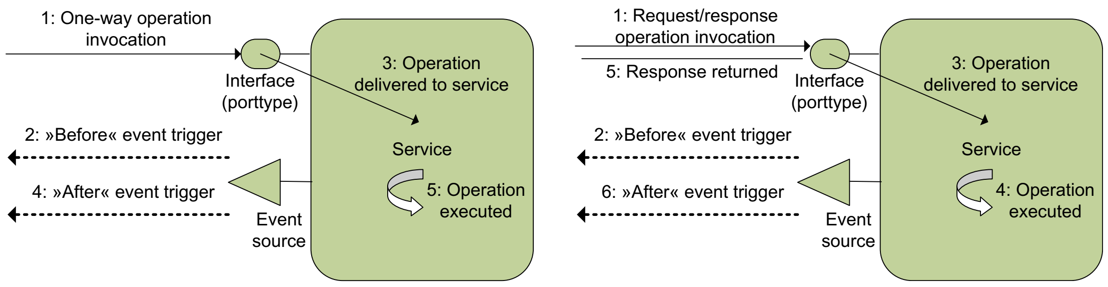

Fig. 5. Sequence for event trigger for one-way and request-response operations.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:wsdlx="http://www.uni-mb.si/wsdl/eventExtensions"
           targetNamespace="http://www.uni-mb.si/wsdl/eventExtensions"
           elementFormDefault="qualified" attributeFormDefault="unqualified">
     <xs:element name="PredefinedEventHeader">
           <xs:complexType>
                 <xs:all>
                       <xs:element name="EventName" type="xs:string"/>
                       <xs:element name="UniqueID" type="xs:string"/>
                       <xs:element name="Version" type="xs:string" minOccurs="0"/>
                       <xs:element name="CreationDateTime" type="xs:dateTime"/>
                       <xs:element name="ValidityFrom" type="xs:dateTime" minOccurs="0"/>
                       <xs:element name="ValidityUntil" type="xs:dateTime" minOccurs="0"/>
                       <xs:element name="ValidityDuration" type="xs:duration" minOccurs="0"/>
                       <xs:element name="ServiceName" type="xs:string"/>
                 </xs:all>
           </xs:complexType>
     </xs:element>
</xs:schema>
```

Listing 1. Obligatory event header schema.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="http://www.uni-mb.si/example"
    xmlns:tns="http://www.uni-mb.si/example"
    xmlns:exs="http://www.uni-mb.si/example/schema"
    xmlns:cbe="http://www.ibm.com/AC/commonbaseevent2_0"
    xmlns:wsdlx="http://www.uni-mb.si/wsdl/eventExtensions">
  ...
  <wsdlx:events>
    <wsdlx:event name="OrderEvent" uid="OrderEventUID" version="1.0">
      <wsdlx:header name="cbe" element="cbe:CommonBaseEvent" use="optional"/>
      <wsdlx:body name="order" element="exs:Order" use="required"/>
      <wsdlx:body name="details" element="exs:ProcessingDetails" use="optional"/>
      <wsdlx:fault name="orderFault" element="exs:OrderFault"/>
      <wsdlx:fault name="processingFault" element="exs:OrderProcessingFault"/>
    </wsdlx:event>
  </wsdlx:events>
  ...
</definitions>
```

Listing 2. Event declaration.

Next, we specify the headers, bodies, and faults that the event consists of. To specify a header or a body, we use the <*wsdlx:header*> or the <*wsdlx:body*> construct, respectively. We specify the *name* of the header/body and the *element* that defines the corresponding schema. We can also specify whether the header/body is required or optional through the *use* attribute. If we do not specify the *use* attribute, the default is *required*. To specify faults we use the <*wsdlx:fault*> construct and specify the *name* and the schema *element*.

This approach makes the proposed solution usable with arbitrary event representation. This can be a custom event representation, or a well-known representation, such as Common Base Event representation [21], WSDM Event Format [22], or any other.

Listing 2 shows an example declaration of the *OrderEvent*. In addition to the obligatory header (as shown in Listing 1) the event declares an optional header named *cbe* and uses the *cbe:CommonBaseEvent* schema representation. It declares a required body named *order* using *exs:Order* schema element and an optional body named *details* using *exs:ProcessingDetails*. It also declares a fault named *orderFault* using *exs:OrderFault* element and a fault named *processingFault* using *exs:OrderProcessingFault* element. Please note that full syntax specification can be found in Appendix A.

5.2. Event sources

An event source can generate a specific type of events. It can also generate fault events. Fault events are used to signal specific exceptional states that have arisen due to business or system exceptions. The declaration of an event source is done within the <*interface*> section of the WSDL 2.0 interface or within the <*portType*> section of the WSDL 1.1 interface. An example using WSDL 2.0 is shown in Listing 3. We can see that the interface *OrderProcessingInt* declares an <*wsdlx:eventSource*> named *OrderProcessed*. To denote that the event is generated, an *output* is specified. It declares that this event source generates events of type *OrderEvent* in the specified version. The event source can also generate a fault (*outfault*) related to the *OrderEvent*. The fault name that can be generated is *orderFault*.

Listing 4 shows the <*wsdlx:eventSource*> declaration for WSDL 1.1. The major difference is that the event source is declared within the <*portType*> section. To follow the WSDL 1.1 syntax we use the *output* to specify the event and the *fault* to specify the related fault.

5.3. Event sinks

An event sink is an event consumer. It can consume a specific event. It can also consume specific faults, related to the event. An event sink can also generate faults. This is particularly useful to signal exceptions related to receiving the events and the exceptions related to the processing of received events.

Often, an event sink might not want to react on all events of the same type. Therefore, we provide the ability to filter events. Events can be filtered using the selected expression language. The default expression language is XPath, however XQuery or a similar language can be used too. An event sink can filter incoming events and the related faults. Particularly when filtering incoming faults

```
<?xml version="1.0" encoding="utf-8"?>
<description …>
  ...
  <interface name="OrderProcessingInt">

    <wsdlx:eventSource name="OrderProcessed">
      <output wsdlx:event="tns:OrderEvent" wsdlx:version="1.0"/>
      <outfault wsdlx:event="tns:OrderEvent" wsdlx:version="1.0"
                wsdlx:name="tns:orderFault"/>
    </wsdlx:eventSource>

  </interface>
  ...
</description>
```

Listing 3. Event source declaration using WSDL 2.0.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions …>
  ...
  <portType name="OrderProcessingInt">

    <wsdlx:eventSource name="OrderProcessed">
      <output wsdlx:event="tns:OrderEvent" wsdlx:version="1.0"/>
      <fault wsdlx:event="tns:OrderEvent" wsdlx:version="1.0"
                wsdlx:name="tns:orderFault"/>
    </wsdlx:eventSource>

  </portType>
  ...
</definitions>
```

Listing 4. Event source declaration using WSDL 1.1.

a certain attention should be put to the filtering expression, as it is not wise to filter out the faults on which the service might need to react.

Event sink can have different behavior patterns. They are specified using the *pattern* attribute with the http://www.wsdlx.org/ns/wsdl/ URI. Patterns include:

- *Subscription*: The event sink subscribes to the event. Multiple event sinks can subscribe to the same event.
- *PrimarySink*: The event sink subscribes to the event as the primary sink. If multiple event sinks are subscribed to the event, the event will be delivered to the primary sink first. All other subscribed sinks should be denoted as *ObserverSink*s.
- *ObserverSink*: The event sink subscribes to the event as the observer sink. It will receive the event after the primary sink.
- *ExclusiveSink*: The event sink is the exclusive subscriber to a specific event. If the event consumer is the exclusive sink then this sink can be the only subscriber to a specific event type. No other event consumer can subscribe to this event type. This is useful for events, which should not be propagated to an arbitrary number of sinks. It is also useful for transforming the event content or structure.

Event sinks are declared within the <*interface*> section of the WSDL 2.0 interface or within the <*portType*> section of the WSDL 1.1 interface. Example in Listing 5 shows an <*wsdlx:eventSink*> declaration within the *OrderProcessingInt* interface. The event sink is called *OrderReceived*, which denotes the event type. The event sink uses the *subscription* pattern. It declares an incoming *OrderEvent* using the <*input*> element. To filter the incoming events, a <*wsdlx:filter*> expression can be used. An expression language of choice is used, default is XPath. The *OrderReceived* event sink also declares an <*infault*> incoming fault of type *orderFault.* Incoming faults can also be filtered in the same way as events. Finally, the event sink can also generate fault events. In our example an <*outfault*> is declared of type *processingFault.* This way the event sink can signal exceptions related to the business event processing.

Listing 6 shows a declaration of an event sink using WSDL 1.1. Similar as in the previous example an *OrderReceived* event sink is declared using the *Subscription* type. This event sink receives the incoming *OrderEvent*s, which can be filtered. The major difference is in the fault declaration. WSDL 1.1 does not support the notion of incoming and outgoing faults. Therefore we propose an optional *wsdlx:direction* attribute, which can have values *"in"* or *"out"*. This way an event sink can specify the direction of the fault events. Incoming fault events can be filtered, while outgoing cannot as it makes no sense to filter outgoing events.

5.4. Event triggers

Often it makes sense to trigger an event upon receiving a method invocation. This is particularly useful for scenarios where we introduce events into solutions that have originally been designed for operation invocation style only. We propose an

```
...
<interface name="OrderProcessingInt">

  <wsdlx:eventSink name="OrderReceived"
                   pattern="http://www.wsdlx.org/ns/wsdl/Subscription">

    <input wsdlx:event="tns:OrderEvent" wsdlx:version="1.0">
      <wsdlx:filter expression="${XPath expression}"/>
    </input>

    <infault wsdlx:event="tns:OrderEvent" wsdlx:version="1.0"
             wsdlx:name="tns:orderFault"/>
    <!-- filter optional -->

    <outfault wsdlx:event="tns:OrderEvent" wsdlx:version="1.0"
              wsdlx:name="tns:processingFault"/>
    <!-- filter n/a to outfault -->

  </wsdlx:eventSink>
</interface>
...
```

Listing 5. Event sink declaration using WSDL 2.0.

```
...
<portType name="OrderProcessingInt">

  <wsdlx:eventSink name="OrderReceived"
                   pattern="http://www.wsdlx.org/ns/wsdl/Subscription">

    <input wsdlx:event="OrderEvent" wsdlx:version="1.0">
      <wsdlx:filter expression="${XPath expression}"/>
    </input>

    <fault wsdlx:direction="in"
           wsdlx:event="OrderFault" wsdlx:version="1.0"
           wsdlx:name="tns:orderFault"/>
    <!-- filter optional -->

    <fault wsdlx:direction="out"
           wsdlx:event="OrderFault" wsdlx:version="1.0"
           wsdlx:name="tns:processingFault"/>
    <!-- filter n/a to outfault -->

  </wsdlx:eventSink>

</portType>
...
```

Listing 6. Event sink declaration using WSDL 1.1.

extension that allows triggering an event declaratively when a client invokes an operation on a service. This allows involving other services into the invocation using a simple WSDL declaration. This way no change in the behavior of the service or the client is required. This is particularly useful in real-world scenarios where we often cannot modify the source code of the service or the client.

To trigger an event based on the operation invocation we propose a <*wsdlx:eventTrigger*> construct that can be used within the <*operation*> declaration. Event can be triggered before the operation invocation is delivered to the service, or after it is delivered. We define this using the *mode* attribute, which can be set to *before* or *after*. In the *before* mode the event is triggered immediately after the operation is invoked on the service interface and before the operation implementation is executed. If the event is triggered *after* the invocation then for one-way operations the event is triggered immediately after the operation invocation is delivered to the operation implementation (before the operation implementation is executed). For request-response operations, the event is triggered after the operation implementation has been executed, and the output message has been generated and returned to the client (Fig. 5).

```
...
<interface name="OrderProcessingInt">
  <fault name = "OrderFault"
          element = "exs:OrderFault"/>

  <operation name="submitOrder"
          pattern="http://www.w3.org/ns/wsdl/in"
          style="http://www.w3.org/ns/wsdl/style/iri">
      <input messageLabel="In"
             element="exs:Order" />
      <outfault ref="tns:OrderFault"
                messageLabel="Out"/>

      <wsdlx:eventTrigger event="tns:OrderEvent"
                          version="1.0"
                          mode="before"
                          transformation="${XSLT}">

         <wsdlx:mapToEvent source="input" transformation="="${XSLT2}"
                           eventPart="header" name="tns:cbe"/>
         <wsdlx:mapToEvent source="input" transformation="="${XSLT3}"
                           eventPart="body" name="tns:order"/>
         <wsdlx:mapToEvent source="outfault" eventPart="fault" name="tns:orderFault"/>

         <wsdlx:filter expression="${XPath}"/>
      </wsdlx:eventTrigger>
  </operation>
</interface>
...
```

Listing 7. Event trigger declaration using WSDL 2.0.

We can make a transformation of the incoming message before we trigger the event. This way we can transform the message payload of the operation into the schema representation used for the event. A transformation can be done using XSLT. It is specified with the *transformation* attribute, which is optional.

We have to specify how the incoming messages of the operation will map to the event. We specify the mapping for *input*, *output*, *infault*, and *outfault* messages using the <*wsdlx:mapToEvent*> construct. For each we specify to which part of the event it maps (header, body, fault) and specify the name of that part. Here we define the XSLT transformation too (using the optional *transformation* attribute on the <*wsdlx:mapToEvent*> construct).

Optionally we can apply a filter, which defines when to trigger an event. This way we can trigger an event only for selected operation invocations, for example if the order amount is larger than a specific value. To apply a filter we use the <*wsdlx:filter*> and specify the expression is the selected expression language (default is XPath).

Listing 7 shows an example of the <*wsdlx:eventTrigger*> declaration in WSDL 2.0 for the *submitOrder* one-way operation. The event *OrderEvent* is generated before the operation is invoked on the service.

Listing 8 shows an example of the same declaration using WSDL 1.1. The major difference is that that here only *input*, *output*, and *fault* messages are supported (WSDL 1.1 does not support *infaults* and *outfaults*).

5.5. Bindings

Event sources, sinks, and triggers can be bound to any protocol, supported by the underlying middleware infrastructure. This can be SOAP, HTTP GET or POST, or a specific binding, such as JMS (Java Message Service). This approach is compliant with the way operations are bound to specific protocol in WSDL. Therefore, we have defined two message exchange patters (MEP) for events:

- *Event sink:* http://www.wsdlx.org/soap/mep/event/sink.
- *Event source:* http://www.wsdlx.org/soap/mep/event/source.

To bind events to a transport protocol, we have two choices. Event can be transported using one-way *fire-and-forget* approach, or an *acknowledge* message can be delivered from the event sink to the event source. We select the approach depending on the desired behavior. We specify the transport approach for event source MEP (event sources and event triggers).

Listing 9 shows the WSDL 2.0 SOAP bindings for the event source, event sink and event triggers.

The WSDL 1.1 bindings are very similar to the WSDL 2.0 bindings hence we do not show them. Events are transferred in an asynchronous manner therefore the need for correlation arises. For events, which are transferred in the *acknowledge* approach, the event and its acknowledgement have to be correlated. Often events have to be correlated among each other. Our approach supports both manual and automatic correlation. Automatic correlation depends on the selected protocol and can be used only with protocols that support correlation. If SOAP is used then automatic correlation is done using WS-Addressing. Manual correlation can use a specific set of event elements. The proposed extensions for events are compatible with other WS specifications, such as WS-Security, which can be used for securing events, event sources and sinks; WS-Addressing for correlation; and WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity for transaction support.

The proposed WSDL extensions for events are programming language independent. Therefore, they can be mapped to any language and use it to implement event services, including Java, C#, C++, etc. In each language, there are several choices how to implement event sources, sinks and triggers. In Java, event sinks can be implemented as POJOs (Plain Old Java Objects), JMS, or MDBs (Message Driven Beans). The event model can also be used in the BPEL, which we will discuss in the next section.

6. Proposed extensions to BPEL

BPEL is an executable language for specifying business processes as orchestrations of services. BPEL originally supports orchestrations based on operation invocations. We propose extensions that add support for event-based service orchestration. Event extensions can raise the level of loose coupling and can make orchestrations more flexible and adaptable to business needs.

We propose specific extensions to BPEL 2.0 to support the notion of business events in BPEL. We introduce new activities for triggering events and for listening to events, and we propose extensions to the event and fault handlers, to variables, and to

```
...
<portType name="OrderProcessingInt">

   <operation name="submitOrder">
      <input message="tns:OrderMsg"/>
      <fault name="OrderIncompleteFault" message="tns:OrderFaultMsg"/>
      <wsdlx:eventTrigger event="tns:OrderEvent"
                          version="1.0"
                          mode="before"
                          transformation="${XSLT}">

         <wsdlx:mapToEvent source="input" transformation="="${XSLT2}"
                           eventPart="header" name="tns:cbe"/>
         <wsdlx:mapToEvent source="input" transformation="="${XSLT3}"
                           eventPart="body" name="tns:order"/>
         <wsdlx:mapToEvent source="fault" eventPart="fault" name="tns:orderFault"/>

         <wsdlx:filter expression="${XPath}"/>
      </wsdlx:eventTrigger>
   </operation>
</portType>
...
```

Listing 8. Event trigger declaration using WSDL 1.1.

```
...
<binding name="OrderProcessBinding"
         interface="tns:OrderProcessingInt"
         type="http://www.w3.org/ns/wsdl/soap"
         wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/">
  <wsdlx:eventSink ref="tns:OrderReceived"
    wsoap:mep="http://www.wsdlx.org/soap/mep/event/sink"/>

  <wsdlx:eventSource ref="tns:OrderProcessed"
    wsoap:mep="http://www.wsdlx.org/soap/mep/event/source"
    wsdlx:approach="http://www.wsdlx.org/soap/approach/event/acknowledge"/>

  <wsdlx:eventTrigger ref="tns:OrderReceived"
    wsoap:mep="http://www.wsdlx.org/soap/mep/event/source"
    wsdlx:approach="http://www.wsdlx.org/soap/approach/event/fire-and-forget"/>

</binding>
...
```

Listing 9. Event bindings using WSDL 2.0.

property aliases. For receiving events we propose a new activity <*bpelx:catchEvent*>. For triggering events we propose a new activity <*bpelx:triggerEvent*>. We propose an extension to the <*pick*> activity to react on events. We also propose an extension to <*variable*> to allow storing the events and an extension to <*vprop:propertyAlias*> to enable using event data in properties for correlation. To catch event faults we propose an extension of the <*catch*> activity within the <*faultHandlers*> section. We also propose an extension of the event handler <*onEvent*> activity to accommodate events. We have also incorporated the usage of events to all other BPEL activities, such as assigns, conditions, and loops. This way we propose a full set of extensions for events in BPEL. Full syntax specification of the extensions is provided in Appendix B.

Originally, BPEL does not support events as described in this article, although it includes event handlers (<*eventHandlers*>). Event handlers in BPEL allow BPEL processes to react on two specific types of events:

- Inbound messages that correspond to a WSDL operation invocation. They are handled within <*onEvent*> part of the event handler; and
- Time-based events (alarms) such as deadlines and durations. They are handled within <*onAlarm*> part of the event handler.

The semantics of the original <*onEvent*> is similar to a <*receive*> activity, in that it waits for the receipt of an inbound message of operation invocation. In the context of events as described in this paper, the <*onEvent*> name is confusing because its purpose is to react on operation invocations (therefore it should be called <*onOperation*> or <*onMessage*>). We propose extensions to original BPEL event handlers and <*onEvent*> activity to enable support for business events in a broader sense that does not require operation invocation.

We have defined the event extensions for executable and abstract BPEL processes. The proposed extensions use the standard BPEL extension mechanism <*extensionActivity*> and can therefore be implemented on any BPEL 2.0 compliant server. For extensions we use the namespace URI http://www.uni-mb.si/bpel/eventExtensions, for which we use the alias *bpelx*.

6.1. Extensions to variables

In BPEL variables are used to store data related to incoming and outgoing messages of operations. We have extended the variables to enable storing the data related to events, as they are defined in WSDL. To achieve this, we have added a *bpelx:eventType* attribute to the variable declaration. We use this attribute to specify the event type that we would like to store in the variable. Listing 10 shows an example of variable declaration *OrderEventVariable*, which is used to store events of type *OrderEvent*.

We can use this new variable type as any other variable. To write queries using XPath or selected query language, we use the following syntax, where we first specify the variable name, then the event part (header, body or fault), then the name of the part and finally the query expression, as shown on Listing 11 (generic form and example).

6.2. Triggering and catching events

To trigger events from BPEL we introduce a new activity <*bpelx:triggerEvent*>. To trigger the event we specify the partner link, the port type, the event source, and the input variable that stores the event data. Optionally we can specify all standard attributes and elements, including correlations, fault handlers, compensation handlers and to and from parts. Full syntax is provided in Appendix B. Example is listed in Listing 12.

To catch (receive) an event, we introduce the <*bpelx:catchEvent*> activity. Similarly as before we have to specify the partner link, the port type, and the event sink (that is used to receive the events). We also specify the variable used to store event data. Example is shown in Listing 13. Optionally we can specify standard attributes and elements, correlations and from parts.

BPEL also allows a process to wait for more than one operation invocation and a time event using the <*pick*> activity. This is done using <*onMessage*> and <*onAlarm*> activities, respectively. To handle events within the <*pick*> activity we have extended the <*pick*> activity with the <*bpelx:onEvent*> branch. The <*bpelx:onEvent*> allows a process to wait for events within the <*pick*> activity. We can specify one or more <*bpelx:onEvent*> branches. For each we specify the partner link, port type, event sink and variable name used to store incoming event data. Optionally we can specify message exchange pattern, correlations and from parts. Listing 14 shows a simple example with one <*bpelx:onEvent*> branches waiting for the event on the event sink *OrderReceived* on the port type *OrderProcessingInt* on partner link *OrderProcessing*. It uses *OrderEventVariable* to store the incoming event data.

```
<variable name="OrderEventVariable"
          bpelx:eventType="tns:OrderEvent" />
```

Listing 10. BPEL variable extension for events.

```
$variable-name.(header|body|fault).part-name/XPath-expression
$OrderEventVariable.header.cbe/cbe:situation/cbe:situationData/cbe:msgDataElement
```

Listing 11. Queries for event variables.

```
<bpelx:triggerEvent partnerLink="OrderProcessing"
                 portType="tns:OrderProcessingInt"
                 eventSource="tns:OrderProcessed"
                 inputVariable="OrderEventVariable"/>
```

Listing 12. Trigger event.

```
<bpelx:catchEvent partnerLink="OrderProcessing"
                 portType="tns:OrderProcessingInt"
                 eventSink="tns:OrderReceived"
                 variable="OrderEventVariable"/>
```

Listing 13. Catch event.

6.3. Event handlers

To enable catching events from BPEL event handler we have introduced an extension to the <*onEvent*> activity. The original <*onEvent*> activity allows a BPEL process to react on an incoming operation invocation, which is handled using <onEvent> activity. It also allows a BPEL process to react on a certain time duration or deadline using <*onAlarm*> activity. To enable reacting on events, we have extended the <*onEvent*> activity. Please notice, that the original <*onEvent*> activity has a misleading name, because its purpose is to react on operation invocations, therefore it should be called <*onOperation*> or <*onMessage*>.

To use the extended <*onEvent*> activity for handling events, we have to specify the partner link, the port type, the event sink. Optionally we can specify the event type that the event handler should react on. If it is not specified the event handler will react on all events defined within an event sink. We also specify a variable that is used to store the incoming event data. Optionally we can specify message exchange, correlations and from parts. Listing 15 shows an example that will handle *OrderEvent* event type on the *OrderReceived* event sink on the *OrderProcessingInt* port type of the *OrderProcessing* partner link and will store the event data into the *OrderEventVariable* variable.

6.4. Fault handlers

To enable handling faults that are thrown from event sources or sinks, we have introduced an extension for the BPEL fault handler. We extend the <*faultHandlers*> <*catch*> activity with adding the *fault event type* on which the fault handler should react. This way the fault handler can react on a specific fault related to the event. Alternatively we can catch event-related faults (and other faults) using the <*catchAll*> activity.

Listing 16 shows an example where the fault handler catches the *OrderProcessingFault* that is related to the *OrderEvent* event type.

6.5. Correlation and property aliases

BPEL processes use a stateful model. Messages sent to the business process instance need to be correlated in order to be delivered to the correct instance of the BPEL process. The same holds true for the events. To enable correlation for events, we have extended the notion of property aliases, through which a developer can specify the correlation properties. Our approach requires only the extension of the property aliases. Developers can use all standard BPEL correlation activities for events in the same way as for correlation with operation invocation messages.

To extend the <*vprop:propertyAlias*> we have added the *bpelx:eventType* and the *bpelx:eventPartType* attributes. The *bpelx:eventType* specifies the event type that we would like to use in the property alias. We specify the name of the event type. The *bpelx:eventPartType* specifies the part of the event, which can be header, body, or fault. We also need to specify the event part name.

```
<pick>
  <bpelx:onEvent partnerLink="OrderProcessing"
            portType="tns:OrderProcessingInt"
            eventSink="tns:OrderReceived"
            variable="OrderEventVariable">
    <!-- Do something -->
  </bpelx:onEvent>
  …
</pick>
```

Listing 14. On event pick extension.

```
<eventHandlers>
  <onEvent partnerLink="OrderProcessing"
          portType="tns:OrderProcessingInt"
          bpelx:eventSink="tns:OrderReceived"
          bpelx:eventType="tns:OrderEvent"
          variable="OrderEventVariable">
      <scope>
          <!-- Do something -->
     </scope>
  </onEvent>
</eventHandlers>
```

Listing 15. Event handler extension.

```
<faultHandlers>
  <catch faultName="tns:OrderProcessingFault"
          bpelx:faultEventType="tns:OrderEvent" >
    <!-- Do something -->
  </catch>
</faultHandlers>
```

Listing 16. Fault handler extension.

Listing 17 shows an example definition of a property alias *OrderEventPropertyAlias*. It uses the *OrderEvent* event type. More specifically, it uses the *header* part with the name *cbe*.

The proposed extensions cover everything that is required to incorporate events in BPEL. Full syntax is described in Appendix B. We can use events in all other BPEL activities implicitly or explicitly. This way BPEL processes can manage, catch, trigger, store and react on events the same way as for operation invocations. With this, we have concluded our discussion on BPEL extensions for events.

7. Proof-of-concept and discussion

To implement the proposed extensions for events, we need to extend the tools, which are used by a specific programming environment for generating source code out of WSDL interfaces. We also need to modify the server for hosting web services and implement support for events. We also have to implement the support for events in the ESB, as described in Section 2. To support BPEL extensions the corresponding process engine has to be extended to support events (and the new BPEL activities).

For the proof-of-concept, we have developed a prototype implementation in Java EE. There are several possibilities how to implement WSDL and BPEL extensions. First possibility is to natively develop a SOA platform with added support for events. This is a complex undertaking. Therefore, we have decided to use and extend open source products. Our goal has been to implement the proposed extensions as a proof-of-concept, therefore the efficiency has not been of top priority. To implement WSDL extensions we have used Apache Axis2 version 1.3 [23] and implemented the WSDL event extensions. We have used the Apache Tomcat to host the services. To implement the bus, we have used JMS, through which we have simulated the event behavior as described in this article. To implement BPEL extensions for events, we have used the open source BPEL engine Apache ODE 1.1 [24].

On the prototype platform, we have implemented three real-world pilot projects, where we have evaluated the usability of the proposed event extensions. We have implemented one pilot project from scratch. For the other two projects, we have started from existing SOA solution that used operation invocations only and have extended it with events. In all three cases, we have figured out that events are a more efficient approach.

The first pilot project was a composite application for reserving company cars. This application was based on a real use case for a company, which had a pool of company cars and employees could reserve the cars for business trips. The requirement was that this application integrates with existing applications, such as the employee evidence application and the car management application. The composite application consisted of a BPEL process, and the services, which exposed the functionality of existing systems (and the user interface).

To be able to measure the effort of the event-driven approach and to compare it to the “classic” operation-invocation approach, two teams have developed the same application. The first team has used the event extensions, as described in this paper. The second team has used the classic SOA/WSPA approach without events.

The development has been done in three iterations. In the first iteration, the relatively simple process has been implemented: the BPEL process checked for the employee status using a service that exposed the functionality of the employee evidence application. Based on the status the employee could reserve different types of

```
  <vprop:propertyAlias propertyName="OrderEventPropertyAlias"
          bpelx:eventType="tns:OrderEvent"
          bpelx:eventPartType="header"
          part="cbe">
    <vprop:query>cbe:CommonBaseEvent/@globalInstanceId</vprop:query>
  </vprop:propertyAlias>
```

Listing 17. Property alias extensions for correlation.

Table 2
Comparison of effort with and w/o event extensions for three use cases.

	Iteration 1			Iteration 2			Iteration 3			Total Effort		
	With events	W/o events	Ratio in %	With events	W/o events	Ratio in %	With events	W/o events	Ratio in %	With events	W/o events	Ratio in %
Pilot project 1	15.50	18.25	17.74	8.50	11.50	35.29	7.00	9.00	28.57	31.00	38.75	25.00
Pilot project 2	n/a	n/a	n/a	52.00	63.00	21.15	21.00	29.00	38.10	73.00	92.00	26.03
Pilot project 3	n/a	n/a	n/a	34.50	43.00	24.64	25.00	37.00	48.00	59.50	80.00	34.45
			17.74			23.68			41.51			28.90

cars (small, medium, full-size). Next, the BPEL process made the reservation, waited for the reservation confirmation, and allowed the employee to fill the report on car usage (after the car has been returned), which the BPEL process submitted to the car management application. To achieve this, a second service exposed the car management application functionalities. In the first iteration, we could observe an advantage of the development effort for the event approach, which was ~18% shorter, as shown in Table 2. This was expected, because both solutions have been built from scratch with well-defined requirements. Event approach allowed less development work for BPEL and WSDL.

In the second iteration, the composite application had to be extended. The BPEL process has been extended to make the calculation of the traveling allowance. It also provided this information to the salary application. At the same time, the work time management application had to be updated with the information that the employee has been out of the office on the business trip. The event-driven approach showed considerable benefits. To include the new services for traveling allowances and work time recording into the BPEL process, the event-driven approach did not require any change to the BPEL process. It only required defining the corresponding event sinks on the new services, which consumed the events that have already been triggered in the BPEL process. Because the BPEL process did not have to change, the complete compile/test/deploy cycle was not necessary. Even more, the event-driven approach allowed us to add the new functionalities step-by-step. When the first service for traveling allowances has been developed it only had to be registered to consume the events. From that moment on it could participate in the composite application. The same holds true for the second service. The team, which used the traditional approach without events, was faced with more difficulties. They had to modify the BPEL process flow in order to invoke the new services for travel allowances and work time recordings, and to repeat the whole compile/test/deploy cycle.

In the third iteration, the requirement was to check the history of car reservations for each employee (in addition to the employee status) and to add integration with the car maintenance part of the car management application. In addition, an audit trail had to be implemented with the complete history log of the operations. In this iteration, the event-driven approach showed similar advantages as in the second iteration, leading to less development effort again. Particularly efficient was the implementation of the audit trail, which only required a definition of event sinks for all events that should be included in the trail. The team that used the approach without events had more problems again. They had to insert an explicit call to the audit trail after each relevant BPEL process activity. Please note that the audit trail was not a simple log, which could be realized by some sort of system logging feature.

The advantage of events has been particularly evident when the audit trail had to be extended to services as well. Here the decision that a service can have an interface (portType) and an event sink and an event source at the same time has proved very useful. This allows the development of business services that can react through various channels in a flexible and loosely coupled way. In our scenario, we used this functionality to add event sources to all services, which needed to be included in the audit trail. Development of self-sustainable services with business functionality is important for sound design and promotes service reuse.

The second pilot project was a procurement process. Procurement process is a complex process with different scenarios depending on the value of the order. It also changes quite often due to the changes in the procurement policies. We started with an implemented procurement process for small order values. This process has been a part of a composite application (similar to the previous pilot project). It has been implemented using traditional SOA approach without events. From here on, two parallel teams have been assigned to extend the process. The first team used the event extensions, while the second team used the traditional approach. The procurement process had to be extended to support medium-value and large-value orders. This basically required to include the invocation of several new services into the BPEL processes (because of the complexity, there were several BPEL processes implemented). The "event team" had a disadvantage at the beginning, because the existing process from the first iteration did not use events at all, therefore the team could not use the same approach as in the first pilot. Rather the existing project had to be modified to include the event sources, sinks, and triggers on the process and the services.

When adding events to the existing SOA solution, we have confirmed that the event triggers, which can be bound to operation invocations on WSDL, are particularly useful. Without events, it is difficult to hook to service invocations and invoke additional services. The only way this can be done without events is with vendor-specific hooks to the service bus. Our event extensions on the other hand provide event triggers, which are a systematic solution that allows triggering events when a service operation is invoked.

We can also apply filters to event triggers. This way we can trigger events only if certain conditions are met. For example, based on the order value we triggered a different event for small, medium, and large order values, respectively. This way we covered the different behavior for small, medium and large amount orders. The important part is that we can do all this declaratively with modifications of WSDL only and without changing any source code of the services. This is important, as it does not require a new test/deployment cycle. It is also very useful in cases where we do not have access to the service source code.

Filters are also useful on event sinks. We have successfully used them to catch only those events that are of value to a specific service. In the procurement pilot, we used the order value filter in most cases. Filtering events on the level of services (and not in the service implementation) makes the interactions more overlookable and limits the event traffic.

Using event triggers and filters the first team, which used event extensions, could demonstrate less effort in the first extension (second iteration) of the procurement process, although not as much as in the first pilot project. However, in the third iteration the advantage over the traditional approach was even higher.

The third pilot project was a customer order process for a telecommunication provider. The process starts with the customer contract for a new cellular subscription and continues with the customer verification/authorization, contract verification, enablement of the cellular subscription, activation of the phone number, and notification to the customer via a short message. The process integrates several existing applications. The initial project has been developed without using events. For the next iterations, we have proceeded with two teams. There have been two iterations: extending the process with additional functionalities, and transforming the process to achieve eTom [83] and NGOSS [84] compliance. For extending the functionality, we had a similar experience as with the previous pilot project. The team that used events has been more productive. This has been emphasized as the process had to be extended to support triple play (Cellular, Internet, TV), and quad play products (Cellular, Internet, TV, Landline phone). Here the process had to integrate with external services, which was much easier to achieve using events.

To achieve compliance with NGOSS, several things had to be done: the process flow had to be restructured, services had to be redesigned to comply with TAM (Telecom Application Map) [85], and service interfaces and data structures had to comply with the SID Information Framework [86]. The team that used the event extensions spent considerably less effort. The main reason has been that the XML representation of events, as proposed in this article, is flexible and extensible. The fact that an event can have multiple headers, bodies, and faults proved to be the right decision, which the team used to adapt the structures to SID. This way they could extend the event header and payload data without the need to modify existing event sources and sinks. A particularly useful feature has proved to be the transformation of the incoming messages before we trigger the events. This allowed to adapt to SID structures step-by-step and use the transformations to access those services, which have not adapted yet. Although some ESBs provide vendor-specific hooks to achieve transformations on the mediation level, our approach proved to be more flexible and could be implemented with less effort, because in our approach transformation can be defined on the services directly, while ESBs require a separate procedure. Our approach is also generic and not bound to a specific ESB.

Our approach also allowed relating various fault messages to an event, which also makes sense, as faults should be related to specific types of events. With our approach, new faults can be added without the need to modify existing event sources and sinks. Our approach also allows to define general faults that are not bound to specific events. In this case, we only define the faults within the event declaration.

We can also report about positive experience with the BPEL event extensions. Orchestrating processes with event-enabled services has proved successful. BPEL extensions have proved to be well integrated with BPEL language and can be flexibly incorporated into the BPEL code and used with all BPEL activities. This was confirmed on all three pilot projects. The ability for a BPEL processes to react on events (act as event sinks) and to trigger events (act as event sources) has opened new aspects of service composition. It has considerably simplified the inclusion of new services into existing flows, which is one of the most common scenarios. We proved this in all three pilot projects. Even on the cases where we had to extend BPEL code without event extensions, we could use event triggers to add support for events in an easy and efficient way.

In all three pilot projects, we also implemented a very common use-cases "where is my process". Users often want to see in which activity a specific process instance is. Please consider that we do not talk here about the BPEL execution trail, but about a more high level, business process perspective. Realizing this with traditional BPEL is very difficult and requires us to add an invoke after each activity that we would like to follow. With event extensions, this becomes trivial as it only requires subscribing to the relevant events. In a similar way, integration with Business Intelligence and Business Activity Monitoring solutions can be much simplified, as well as logging, tracing, etc.

The proposed event extensions for WSDL and BPEL have had positive effects on key service design principles [7]. They improved loose coupling because they reduced the dependencies between services. The service consumer does not depend on the service interface and operation signature, but is subscribed to an event or a set of events. As the service only exposes the events, it can better hide the underlying service details, which improves service abstraction. With the ability to compose services using events, the service composability becomes more flexible. It is much easier to include a new event-based service into a composition or modify an existing composition, as it only requires subscription of event-enabled service to the particular event. In contrast, services that are based on the operation invocation require a modification of the composition (BPEL flow).

To assess the development effort and compare the differences between both approaches (with and without event extensions), we measured the development and maintenance effort in person-days. We measured the effort for the development of the business layer (BPEL, services, etc.). In all three cases, both teams have been comparably skilled and had similar experience. Table 2 shows the development and maintenance effort for all three pilot projects. The first iteration in pilot project 1 required 15,5 person-days using event extensions and 18,25 person-days without event extensions. We can observe an 18% reduction of the development time. The second iteration of pilot project 1 showed a $\sim$35% reduction in development time when using event extensions. The third iteration showed a $\sim$29% reduction. Total development effort for the pilot project 1 has been reduced by $\sim$25% with event extensions.

For the pilot project 2, the first iteration has been developed without events, as explained earlier in this section. This is why we have not included the numbers for the first iteration. However, on this pilot project we were able to compare a common scenario, where an existing SOA application has to be extended. We have observed a $\sim$21% reduction of development time in the second iteration of pilot project 2. This is lower than in the first pilot project. The reason is that in the first iteration this project did not use the event extensions, therefore events had to be added in the second iteration, which required additional time. In the third iteration we have observed a $\sim$38% shorter time. The reason is related to the specific requirements, including the requirement to implement audit trail, which is particularly well suited for event-driven approach.

For the pilot project 3, we were faced with a similar situation as with pilot project 2. The first iteration has been developed without event extensions. In the second iteration the event team had to add event support, therefore the observed reduction of time has been $\sim$25%. In the third iteration however we have observed a $\sim$48% reduction of development time for the team using event extensions. The high number is again related to the specific requirements. Modifying the data structures to comply with SID and restructuring the services and processes to comply with TAM and eTom is again well suited for event-driven approach. Such scenarios are however quite common in real-world, therefore we believe that the results of the effort comparison are representative. The proposed WSDL and BPEL extensions for events reduced the overall development and maintenance times considerably. The events made the changes less complex and therefore easier to master, which in our cases has simplified release planning [25], and improved the timeliness of delivery, which is an important success indicator for software projects [26].

The usability of the proposed event extensions would be further improved if support in the development tools would be implemented. Currently we work on an Eclipse plug-in for WSDL. In the future, we will most likely implement an Eclipse BPEL plug-in with added event extensions. Nevertheless, even without this, the prototype implementation has shown that the proposed extensions make sense and that combining SOA with EDA makes sense particularly in business applications. It simplifies the development and maintenance of composite applications. Our approach has been designed with the objective to integrate as seamlessly as possible with WSDL and BPEL.

In addition to several positive aspects, we have to be aware that events are not meant to replace all interactions between services. If used too much, events can lead to a situation, where we lose track of the dependencies between services and processes. Therefore, careful design of event interactions is important. Tracking event dependencies could be simplified with a management tool that would provide run-time analysis, which will be addressed in our future research.

8. Related work

Combination of SOA and EDA concepts with WSDL and BPEL extensions have not been addressed yet in a way comparable to the approach proposed in this paper. Several authors have identified the fact that a combination of SOA and EDA makes sense in business applications. Woods and Mattern [27] have identified the potential of combining SOA and EDA. Taylor et al. [28] have provided a good insight into EDA and SOA and their core components. van Hoof [29] has explained the differences between SOA and EDA, when to use the one or the other and how to combine them. Michelson [30] has given an overview of EDA and the distinction between EDA and CEP. Sholler [31] has explained key issues for SOA, EDA, and WOA (Web Oriented Architecture). Natis and Schulte [32] have compared SOA and EDA from the perspective of business benefits. They all have identified the potential of combining SOA and EDA, but none of them has proposed specific solutions or extensions to EDA or SOA to achieve this. Our approach is complementary to the publish/subscribe communication paradigm, as described by Eugster et al. [33].

Marechaux [65] has come to a similar conclusion as we, that SOA and EDA are two different paradigms that address complex integration challenges. He has proposed the Enterprise Service Bus as an architectural pattern to incorporate event services for detection, triggering and distribution of events. However, Marechaux [65] has not considered extending services and their interfaces with event support. We propose specific extensions to WSDL and BPEL to support events.

Laliwala and Chaudhary [66] have also concluded that existing service-oriented computing standards and technologies are not fully supporting event-driven business process. They propose Event-driven Service Oriented Architecture, which is based on web services, Semantic web and grid computing standards and technologies. Their goal has been to automate the event-driven business process with interoperable integration of scattered services and resources. The approach in [66] is completely different than our approach, and is based on event manager, composition engine, ontology server, rules engine, execution engine, business services, and grid services. Our approach proposed extensions to WSDL and BPEL.

Wieland et al. [67] have identified that SOA and EDA are unique architectural styles widely used in today's industries; however, they mostly exist as isolated systems. They have introduced SOEDA, a development method for workflow-based applications based on event-driven service-oriented architectures. Because of no native support for events in BPEL, SOEDA maps events (from the EPC specification) to message receive activities (of the abstract BPEL). [67] is complementary to our work. We do not focus on the development method, but rather propose extensions to WSDL and BPEL to accommodate events, which could be used by SOEDA to simplify the development method.

An attempt to introduce events to web services/WSPA has been done in the WS-Eventing specification [34]. WS-Eventing [34] defines an eventing service with a set of operations that allow web services to provide asynchronous notifications to interested parties. WS-Eventing defines the simplest level of web services interfaces for notification producers and notification consumers including standard message exchanges to be implemented by service providers that wish to act in these roles, along with operational requirements expected of them. In contrast to our proposal, WS-Eventing does not extend WSDL (nor BPEL) with notion of events, but rather defines a set of operations to subscribe, renew, unsubscribe and get subscription status to mimic events. WS-Eventing also does not address some advanced features, which we have proposed, such as event sink patterns, event triggers, support for transformations, and message exchange patters for events.

Web Services Notification (WS-Notification) consists of three specifications through which web services can disseminate notifications. These OASIS specifications are WS-BaseNotification [35], WS-Topics [36], and WS-BrokeredNotification [37]. WS-BaseNotification defines basic interactions between notification producers and notification consumers. WS-BrokeredNotification defines interfaces for notification brokers. WS-Topic defines a hierarchical topic space. The WS-Notification specifications are related with the WS-Resource framework (WSRF). WSRF is a framework for managing Grid resources through web services and replaces the OGSI (Open Grid Services Infrastructure) specification [38]. WS-Notification defines operations for subscribe, unsubscribe and renew subscription to notifications, to pause/resume subscription and to get current message. Through WSRF it also provides getResourceProperties and TerminationNotification. In contrast to our proposal it does not define extensions to WSDL or BPEL, but rather uses operations to mimic the publish/subscribe pattern. Our approach provides a more comprehensive and a more integrated solution for business events. WS-Notification specifications have on the other hand been designed for general notifications. Our approach also provides support for different event patterns, and event triggers. The notion of event sources and event sinks in our approach is more comprehensive than notification subscriptions in WS-Notification.

An overview of WS-Notification has been done in [39]. WS-Notification is a successor to WS-Events [40] proposed by HP. Huang and Gannon [41] have done a comprehensive comparison of WS-Notification and WS-Eventing. De Labey and Steegmans [42] have extended WS-Notification specification with the notification broker, which supports event correlation. Our approach also supports event correlation, but uses a completely different approach. It does not require a notification broker, but uses WS-Addressing for automatic correlation and correlation properties for manual correlation. Vinoski has discussed the role of notifications and the related specifications in web services in

[43] and [44]. Pallickara and Fox [45] have done an analysis of notification related specifications for web services and Grid applications. In a joint white paper from HP, IBM, Intel, and Microsoft Cline et al. [46] have propose a common set of specifications for resources, events, and management called WS-EventNotification, that will integrate functions from WS-Notification with WS-Eventing.

Notifications are also addressed by the Open Grid Services Infrastructure (OGSI) specification [38]. OGSI objective is the coordination of computing resources across the Internet. OGSI provides the ability for various Grid resources to provide a uniform interface to the OGSA (Open Grid Services Architecture). Although the focus of OGSI is very different to our proposal, OGSI also uses an extension of WSDL to define services interfaces. The OGSI extension to WSDL is much simpler than our extension.

Prior to web services and SOA, other open architectures have introduced event and notification related specifications. CORBA (Common Object Request Broker Architecture), developed by the OMG (Object Management Group), has defined Event Service specification [47] and Notification Service specification [48]. CORBA Event Service supports asynchronous communications between event suppliers and consumers. It defines a mechanism for event propagation through event channels. It does not address event filtering. Similarly as WS-Eventing it prescribes a specific interface for subscribing and unsubscribing to events. In contrast to our proposal, it does not introduce specific extensions to interfaces (the Interface Definition Language). It also does not address the event structure, various event patterns, triggers, and transformations. CORBA Notification Service enhances the CORBA Event Service and introduces structured events, which define data structure to map a generic event to a well-structured event. It introduces event filtering, which is done through filter object and not on the interface directly as in our proposal. It uses a fixed trader constraint language, which is less flexible than our approach, where the filter language can be any expression language, such as XPath or XQuery. Java Message Service (JMS) [49] is a specification for message-oriented middleware that allows Java applications to create, send, and receive a messages in an asynchronous manner. JMS is limited to asynchronous publish/subscribe communication and does not support the notion of interfaces. This way it does not provide the integration between operation invocation and events, as in our approach. It also lacks advanced features, such as filtering and patterns.

We can see that none of the related work has proposed the combination of SOA and EDA in a way where it would extend WSDL and introduce events as native artifacts, which makes them as easy and flexible to use as operations. There have however been several proposals to extend WSDL in other areas, such as extensions for security [50], QoS [51], semantic annotations [79], performance [77], testing [78], and versioning [52], which are complementary to our extensions.

There have also been no attempts to extend BPEL with events. There have been however several proposals to extend BPEL in other areas, such as BPEL Extensions for Sub-Processes [53], BPEL Extension for People (BPEL4People) [54], AO4BPEL for aspect-oriented extensions to BPEL [55], AdaptiveBPEL for development of differentiated and adaptive BPEL processes [56], C-BPEL for incorporating context information [57], BPEL4Chor for modeling choreographies [58], WS-BPEL Extensions for Versioning [59], BPEL for modeling resource-oriented state machines [73], BPEL for REST [74,75], and Ode extensions to BPEL, including as implicit correlations, activity failure and recovery, extension activities and extensible assign operations, external variables, and headers handling [76]. Our BPEL extensions for events are complementary to these extensions.

Events are also addressed in business process modeling languages, such as Event-driven Process Chains [60] and Business Process Modeling Notation (BPMN) [61]. Therefore, it is even more important that events are supported in the Business Process Execution Language to bridge the models across the lifecycle [62]. The proposed BPEL extensions for events allow better alignment of process models with their executable representation. Mapping of BPMN events to BPEL is complex, as BPMN events map to several BPEL constructs, such as receive, invoke, and pick [61]. Translation to BPEL potentially needs to map certain dedicated event subtypes within BPMN to a single event type of BPEL (for instance, external events) [80]. This complicates round-tripping from BPMN to BPEL and back to BPMN. With the proposed BPEL extensions for events, the mapping of BPMN events (and round-tripping) can be simplified considerably.

Barros et al. [81] have expressed common eventing scenarios in business processes. Their assessment has shown that only few of these scenarios are supported in BPMN and BPEL. Decker et al. [82] have proposed a graphical language for modeling composite events in business processes, BEMN (Business Event Modeling Notation). Our BPEL extensions for events are complementary with their work and BEMN models can be translated to BPEL extended with events.

9. Conclusion

In this article, we have proposed a solution for combining the Event Driven Architecture principles with the Service Oriented Architecture principles. We have presented extensions to WSDL and BPEL for events. Our solution introduces events as first-class citizens to SOA/WSPA. It allows services to act as event producers and/or event consumers, but at the same time retain the interfaces (portTypes) and their operations. This way we have successfully combined the event-driven semantics with operation-invocation semantics and have provided developers to choose the most appropriate alternative. We have introduced a flexible and extensible XML representation of events and their payloads. We have proposed specific extensions to WSDL 2.0 and 1.1 to define event sinks and event sources for services. We have also introduced event triggers and enabled combining operation invocations with event triggers to simplify the introduction of events into existing SOA solutions. Furthermore, we have introduced event sink patterns, event filters, event transformations, and message exchange patterns for events.

To enable event-based service orchestrations in business processes, we have proposed extensions to BPEL 2.0. We have introduced new BPEL activities for triggering and catching events. We have added an event handler to the pick activity and enabled a BPEL process to react to events in addition to operation invocation and alarms. We have extended the event handlers and added an event-specific catch activity to fault handlers. To store event data, we have extended the notion of BPEL variables, which can now also store event data in addition to operation messages, elements, and types. To enable event correlation we have extended the notion of properties and property aliases. We have also incorporated the usage of events to all other BPEL activities, such as assigns, conditions, and loops.

The extensions that we have proposed have been designed to work seamlessly with WS specifications, such as WS-Security, WS-Addressing, and WS-Coordination, WS-AtomicTransaction, and WS-BusinessActivity. To demonstrate the usability, we have implemented a prototype implementation of the event extensions. We have developed the extensions upon existing open source products, Apache Axis2 and ODE. We have also tested the event extensions on three pilot projects. The event extensions have positive effects on key service design principles [7], including improved loose coupling, better service abstraction, improved service reuse, and more flexible service composability.

Appendix A. WSDL extensions syntax specifications

Event declaration (same for WSDL 2.0 and WSDL 1.1):

```
<wsdlx:events>
  <wsdlx:import namespace="anyURI" location="anyURI"/>?
  <wsdlx:event name="QName" uid="QName"? version="string"?>*
    <wsdlx:header name="QName" element="NCName" use="required|optional"?/>*
    <wsdlx:body name="QName" element="NCName" use="required|optional"?/>*
    <wsdlx:fault name="QName" element="NCName"/>*
  </wsdlx:event>
</wsdlx:events>
```

Event source declaration in WSDL 2.0:

```
<wsdlx:eventSource name="QName">
  <output wsdlx:event="NCName" wsdlx:uid="QName"? wsdlx:version="string"?/>
  <outfault wsdlx:event="NCName" wsdlx:uid="QName"? wsdlx:version="string"?
            wsdlx:name="NCName"/>*
</wsdlx:eventSource>
```

Event source declaration in WSDL 1.1:

```
<wsdlx:eventSource name="QName">
  <output wsdlx:event="NCName" wsdlx:uid="QName"? wsdlx:version="string"/>
  <fault wsdlx:event="NCName" wsdlx:uid="QName"? wsdlx:version="1.0"?
            wsdlx:name="NCName"/>*
</wsdlx:eventSource>
```

Event sink declaration in WSDL 2.0:

```
<wsdlx:eventSink name="QName"
                 pattern="http://www.wsdlx.org/ns/wsdl/Subscription|
                          http://www.wsdlx.org/ns/wsdl/ExclusiveSink|
                          http://www.wsdlx.org/ns/wsdl/PrimarySink|
                          http://www.wsdlx.org/ns/wsdl/ObserverSink">

  <input wsdlx:event="NCName" wsdlx:uid="QName"? wsdlx:version="string"?>
    <wsdlx:filter expression="expression"/>*
  </input>

  <infault wsdlx:event="NCName" wsdlx:uid="QName"? wsdlx:version="string"?
           wsdlx:name="NCName">*
    <wsdlx:filter expression="expression"/>*
  </infault>

  <outfault wsdlx:event="NCName" wsdlx:uid="QName"? wsdlx:version="string"?
            wsdlx:name="NCName"/>*

</wsdlx:eventSink>
```

Event sink declaration in WSDL 1.1:

```
<wsdlx:eventSink name="QName"
                 pattern="http://www.wsdlx.org/ns/wsdl/Subscription|
                          http://www.wsdlx.org/ns/wsdl/ExclusiveSink|
                          http://www.wsdlx.org/ns/wsdl/PrimarySink|
                          http://www.wsdlx.org/ns/wsdl/ObserverSink">

  <input wsdlx:event="NCName" wsdlx:uid="QName"? wsdlx:version="string"?>
    <wsdlx:filter expression="expression"/>*
  </input>

  <fault wsdlx:direction="in"
         wsdlx:event="NCName" wsdlx:uid="QName"? wsdlx:version="string"?
         wsdlx:name="NCName">*
    <wsdlx:filter expression="expression"/>*
  </fault>

  <fault wsdlx:direction="out"
         wsdlx:event="NCName" wsdlx:uid="QName"? wsdlx:version="string"?
         wsdlx:name="NCName"/>*

</wsdlx:eventSink>
```

Event trigger declaration in WSDL 2.0:

```
<wsdlx:eventTrigger event="NCName" uid="QName"? wsdlx:version="string"?
                    mode="before|after"
                    transformation="anyURI"?>

   <wsdlx:mapToEvent source="input|output|infault|outfault"
                     transformation="anyURI"?
                     eventPart="header|body|fault"
                     name="NCName"/>+

   <wsdlx:filter expression="expression"/>*
</wsdlx:eventTrigger>
```

Event trigger declaration in WSDL 1.1:

```
<wsdlx:eventTrigger event="NCName" uid="QName"? wsdlx:version="string"?
                    mode="before|after"
                    transformation="anyURI"?>

   <wsdlx:mapToEvent source="input|output|fault"
                     transformation="anyURI"?
                     eventPart="header|body|fault"
                     name="NCName"/>+

   <wsdlx:filter expression="expression"/>*
</wsdlx:eventTrigger>
```

Event bindings (same for WSDL 2.0 and 1.1):

```
<wsdlx:eventSink ref="NCName"
  wsoap:mep="http://www.wsdlx.org/soap/mep/event/sink"/>

<wsdlx:eventSource ref="NCName"
  wsoap:mep="http://www.wsdlx.org/soap/mep/event/source"
  wsdlx:approach="http://www.wsdlx.org/soap/approach/event/acknowledge|
                 http://www.wsdlx.org/soap/approach/event/fire-and-forget"/>

<wsdlx:eventTrigger ref="NCName"
  wsoap:mep="http://www.wsdlx.org/soap/mep/event/source"
  wsdlx:approach="http://www.wsdlx.org/soap/approach/event/acknowledge|
                 http://www.wsdlx.org/soap/approach/event/fire-and-forget"/>
```

Appendix B. BPEL extensions syntax specifications

<bpelx:catchEvent> for receiving events:

```
<bpelx:catchEvent partnerLink="NCName"
                  portType="QName"?
                  eventSink="NCName"
                  variable="BPELVariableName"?
                  createInstance="yes|no"?
                  standard-attributes>
   standard-elements
   <correlations>?
     <correlation set="NCName" initiate="yes|join|no"? />+
   </correlations>
   <fromParts>?
     <fromPart part="NCName" toVariable="BPELVariableName" />+
   </fromParts>
</bpelx:catchEvent>
```

<bpelx:triggerEvent> for triggering events:

```
<bpelx:triggerEvent partnerLink="NCName"
                  portType="QName"?
                  eventSource="NCName"
                  inputVariable="BPELVariableName"?
                  standard-attributes>
   standard-elements
   <correlations>?
     <correlation set="NCName" initiate="yes|join|no"?
                  pattern="triggerEvent|catchEvent|acknowledge"? />+
   </correlations>
   <catch faultName="QName"?
          faultVariable="BPELVariableName"?
          bpelx:faultEventType="QName"?
          faultElement="QName"?>*
     activity
   </catch>
   <catchAll>?
     activity
   </catchAll>
   <compensationHandler>?
     activity
   </compensationHandler>
   <toParts>?
     <toPart part="NCName" fromVariable="BPELVariableName" />+
   </toParts>
   <fromParts>?
     <fromPart part="NCName" toVariable="BPELVariableName" />+
   </fromParts>
</bpelx:triggerEvent>
```

<bpelx:onEvent> within <pick> for receiving events:

```
<pick createInstance="yes|no"? standard-attributes>
  standard-elements
  <bpelx:onEvent partnerLink="NCName"
           portType="QName"?
           eventSink="NCName"
           variable="BPELVariableName"?>+
    <correlations>?
      <correlation set="NCName" initiate="yes|join|no"? />+
    </correlations>
    <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableName" />+
    </fromParts>
    activity
  </bpelx:onEvent>
</pick>
```

<variable> extension for events:

```
<variables>
  <variable name="BPELVariableName"
         messageType="QName"?
         bpelx:eventType="QName"?
         type="QName"?
         element="QName"?>+
    from-spec?
  </variable>
</variables>
```

<vprop:propertyAlias> extension for events:

```
<wsdl:definitions name="NCName" ...>
  <vprop:propertyAlias propertyName="QName"
         messageType="QName"?
         bpelx:eventType="QName"?
         bpelx:eventPartType="header|body|fault"?
         part="NCName"?
         type="QName"?
         element="QName"?>
    <vprop:query queryLanguage="anyURI"?>?
      queryContent
    </vprop:query>
  </vprop:propertyAlias>
  ...
</wsdl:definitions>
```

Fault handler <catch> extension:

```
<faultHandlers>
  <catch faultName="QName"?
       faultVariable="BPELVariableName"?
       ( faultMessageType="QName" | faultElement="QName" | bpelx:faultEventType="QName")? >*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
</faultHandlers>
```

Event handler <onEvent> extension:

```
<eventHandlers>?
  <onEvent partnerLink="NCName"
         portType="QName"?
         ((operation="NCName"
         ( messageType="QName" | element="QName" )?)  |
         (bpelx:eventSink="NCName"
         bpelx:eventType="QName"? ))
         variable="BPELVariableName"?
         messageExchange="NCName"?>*
      <correlations>?
        <correlation set="NCName" initiate="yes|join|no"? />+
      </correlations>
      <fromParts>?
        <fromPart part="NCName" toVariable="BPELVariableName" />+
      </fromParts>
      <scope ...>...</scope>
  </onEvent>
</eventHandlers>
```

References

[1] Y.V. Natis, R.W. Schulte, Business component architecture unites services and events, Gartner Research (2004).
[2] B.J. Nelson, Remote Procedure Call, Ph.D. Dissertation, Computer Science Department, Carnegie-Mellon University, CMU Report CMU-CS-81-119 and Xerox PARC Report CSL-81-9, 1981.
[3] The Open Group, DCE Portal, 2009. <http://www.opengroup.org/dce/>.
[4] Object Management Group, Catalog of OMG CORBA/IIOP Specifications, 2009. <http://www.omg.org/technology/documents/corba_spec_catalog.htm>.
[5] M. Völter, A. Schmid, E. Wolff, Server Component Patterns: Component Infrastructures Illustrated with EJB, Wiley, 2002.
[6] T. Erl, Service-Oriented Architecture (SOA): Concepts, Technology, and Design, Prentice Hall, 2005.
[7] T. Erl, SOA Principles of Service Design, Prentice Hall, 2007.
[8] M. Morisio, C.B. Seaman, V.R. Basili, A.T. Parra, S.E. Kraft, S.E. Condon, COTS-based software development: processes and open issues, Journal of Systems and Software 61 (3) (2002) 189–199.
[9] K.M. Chandy, Event-Driven Applications: Costs, Benefits and Design Approaches, California Institute of Technology, 2006.

[10] G. Hohpe, B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley Professional, 2003.
[11] Oracle, Complex Event Processing in the Real World, White Paper, 2007. <http://www.oracle.com/technologies/soa/docs/oracle-complex-event-processing.pdf>.
[12] G. Mühl, L. Fiege, P. Pietzuch, Distributed Event-Based Systems, Springer-Verlag, New York, Inc., 2006.
[13] C. Wohlin, M. Höst, P. Runeson, A. Wesslén, Software Reliability, Encyclopedia of Physical Sciences and Technology, third ed., vol. 15, Academic Press, 2001.
[14] D. Jordan, J. Evdemon, Web Services Business Process Execution Language Version 2.0, OASIS Standard, 2007. <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>.
[15] W3C, Web Services Description Language (WSDL) 1.1, W3C Note, 2001. <http://www.w3.org/TR/wsdl>.
[16] W3C, Web Services Description Language (WSDL) Version 2.0, W3C Recommendation, 2007. <http://www.w3.org/TR/wsdl20/>.
[17] R. Salz, WSDL 2: Just Say No, O'Rielly xml.com, 2004. <http://webservices.xml.com/pub/a/ws/2004/11/17/salz.html>.
[18] M. Little, Does WSDL 2.0 Matter, InfoQ, 2007. <http://www.infoq.com/news/2007/01/wsdl-2-importance>.
[19] M.B. Juric, K. Pant, Business Process Driven SOA using BPMN and BPEL, Packt Publishing, Birmingham, 2008.
[20] M.B. Juric, B. Mathew, P. Sarang, Business Process Execution Language for Web Services, second ed., Packt Publishing, Birmingham, 2006.
[21] D. Ogle, H. Kreger, A. Salahshour, J. Cornpropst, E. Labadie, M. Chessell, B. Horn, J. Gerken, Canonical Situation Data Format: The Common Base Event, OASIS, 2003. <http://www.oasis-open.org/committees/download.php/3862/commonbase%2520event_situationdata_v20.doc/>.
[22] WSDM Technical Committee, OASIS Web Services Distributed Management (WSDM) Standard, OASIS, 2006. <http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm>.
[23] Apache Software Foundation, Apache Axis2 version 1.3, 2008. <http://ws.apache.org/axis2/download.cgi>.
[24] Apache Software Foundation, Apache ODE, 2010. <http://ode.apache.org/>.
[25] G. Ruhe, M.O. Saliu, The art and science of software release planning, IEEE Software (2005) 47–53.
[26] C. Wohlin, A. von Mayrhauser, M. Höst, B. Regnell, Subjective evaluation as a tool for learning from software project success, Information and Software Technology 42 (14) (2000) 983–992.
[27] D. Woods, T. Mattern, Enterprise SOA: Designing IT for Business Innovation, O'Reilly Media, 2006.
[28] H. Taylor, A. Yochem, L. Phillips, F. Martinez, Event-Driven Architecture: How SOA Enables the Real-Time Enterprise, Addison-Wesley Professional, 2009.
[29] J. van Hoof, How EDA Extends SOA and Why It Is Important, 2006. <http://soa-eda.blogspot.com>.
[30] B.M. Michelson, Event-Driven Architecture Overview, Patricia Seybold Group, 2006. <http://dx.doi.org/10.1571/bda2-2-06cc>.
[31] D. Sholler, Key Issues for SOA, EDA and WOA, Gartner Research (2009).
[32] Y.V. Natis, R.W. Schulte, Advanced SOA for advanced enterprise projects, Gartner Research (2006).
[33] P.T. Eugster, P. Felber, et al., The many faces of publish/subscribe, ACM Computing Surveys 35 (2003).
[34] D. Box, L.F. Cabrera, C. Critchley, F. Curbera, et al., Web Services Eventing (WS-Eventing), W3C Member Submission, 2006. <http://www.w3.org/Submission/WS-Eventing/>.
[35] S. Graham, D. Hull, B. Murray, Web Services Base Notification 1.3 (WS-BaseNotification), OASIS Standard, 2006. <http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf>.
[36] W. Vambenepe, S. Graham, P. Niblett, Web Services Topics 1.3 (WS-Topics), OASIS Standard, 2006. >http://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-os.pdf>.
[37] D. Chappell, L. Liu, Web Services Brokered Notification 1.3 (WS-BrokeredNotification), OASIS Standard, 2006. <http://docs.oasis-open.org/wsn/wsn-ws_brokered_notification-1.3-spec-os.pdf>.
[38] S. Tuecke, I. Foster, et al., Open Grid Services Infrastructure, 2003. <http://www-unix.globus.org/toolkit/draft-ggf-ogsigridservice-33_2003-06-27.pdf>.
[39] P. Niblett, S. Graham, Events and service-oriented architecture: the OASIS web services notification specifications, IBM Systems Journal 44 (4) (2005) 869–886.
[40] N. Catania et al., Web Services Events (WS-Events) Version 2.0, 2003. <http://devresource.hp.com/drc/specifications/wsmf/WSEvents.pdf>.
[41] Y. Huang, D. Gannon, A comparative study of web services-based event notification specifications, in: Proceedings of the 2006 International Conference on Parallel Processing Workshops (ICPPW'06), 2006.
[42] S. De Labey, E. Steegmans, Extending WS-Notification with an expressive event notification broker, IEEE International Conference on Web Services (2008) 312–319.
[43] S. Vinoski, Web services notifications, IEEE Internet Computing 8 (2) (2004).
[44] S. Vinoski, More web services notifications, IEEE Internet Computing 8 (3) (2004).
[45] S. Pallickara, G. Fox, An analysis of notification related specifications for web/grid applications, in: International Conference on Information Technology: Coding and Computing (ITCC'05), 2005.
[46] K. Cline, J. Cohen, et al., Toward Converging Web Service Standards for Resources, Events, and Management, 2006. <http://download.boulder.ibm.com/ibmdl/pub/software/dw/webservices/Harmonization_Roadmap.pdf>.
[47] OMG, CORBA Event Service Specification, Version 1.2, 2004. <http://www.omg.org/docs/formal/04-10-02.pdf>.
[48] OMG, CORBA Notification Service Specification, Version 1.1, 2004. <http://www.omg.org/cgi-bin/doc?formal/04-10-13.pdf>.
[49] M. Hapner, R. Burridge, R. Sharma, J. Fialli, K. Stout, Java Message Service, Version 1.1, Sun Microsystems, 2002.
[50] C. Adams, S. Boeyen, UDDI and WSDL extensions for web service: a security framework, in: Proceedings of the 2002 ACM Workshop on XML Security, 2002, pp. 30–35.
[51] A. D'Ambrogio, A model-driven WSDL extension for describing the QoS of web services, in: Proceedings of the IEEE International Conference on Web Services, 2006, pp. 789–796.
[52] M.B. Juric, A. Sasa, B. Brumen, I. Rozman, WSDL and UDDI extensions for version support in web services, Journal of Systems and Software (2009), doi:10.1016/j.jss.2009.03.001.
[53] M. Kloppmann et al., WS-BPEL extension for sub-processes – BPEL-SPE, in: A Joint White Paper by IBM and SAP, September, 2005. <http://download.boulder.ibm.com/ibmdl/pub/software/dw/webservices/ws-bpelsubproc/ws-bpelsubproc.pdf>.
[54] A. Agrawal et al., WS-BPEL Extension for People (BPEL4People), Version 1.0, 2007. <http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/BPEL4People_v1.pdf>.
[55] A. Charfi, M. Mezini, Aspect-oriented web service composition with AO4BPEL, in: ECOWS, LNCS, vol. 3250, Springer, 2004, pp. 168–182.
[56] A. Erradi, P. Maheshwari, AdaptiveBPEL: a policy-driven middleware for flexible web services compositions, in: Proceedings of Middleware for Web Services (MWS), 2005.
[57] C. Ghedira, H. Mezni, Through personalized web service composition specification: from BPEL to C-BPEL, Electronic Notes in Theoretical Computer Science 146 (1) (2006) 117–132.
[58] G. Decker, O. Kopp, F. Leymann, M. Weske, BPEL4Chor: extending BPEL for modeling choreographies, in: ICWS, IEEE International Conference on Web Services, 2007, pp. 296–303.
[59] M.B. Juric, A. Sasa, I. Rozman, WS-BPEL extensions for versioning, Information and Software Technology 51 (8) (2009) 1261–1274.
[60] W.M.P. van der Aalst, Formalization and verification of event-driven process chains, Information and Software Technology 41 (10) (1999) 639–650.
[61] OMG, Business Process Modeling Notation (BPMN), Version 1.2, 2009. <http://www.omg.org/docs/formal/09-01-03.pdf>.
[62] N. Medvidovic, P. Grünbacher, A. Egyed, B.W. Boehm, Bridging models across the software lifecycle, Journal of Systems and Software 68 (3) (2003) 199–215.
[63] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D.F. Ferguson, Web Services Platform Architecture, Prentice Hall, 2005.
[64] J. Nitsche, T. van Lessen, F. Leymann, WSDL 2.0 Message exchange patterns: limitations and opportunities, in: ICIW, The Third International Conference on Internet and Web Applications and Services, 2008, pp. 168–173.
[65] J.L. Marechaux, Combining Service-Oriented Architecture and Event-Driven Architecture using an Enterprise Service Bus, IBM Developer Works, 2006. <http://www.ibm.com/developerworks/webservices/library/ws-soa-eda-esb/index.html>.
[66] Z. Laliwala, S. Chaudhary, Event-driven service-oriented architecture, in: International Conference on Service Systems and Service Management, ICSSSM, IEEE Computer Society, 2008, pp. 1–6.
[67] M. Wieland, D. Martin, O. Kopp, F. Leymann, SOEDA: A methodology for specification and implementation of applications on a service-oriented event-driven architecture, in: Proceedings of the 12th International Conference on Business Information Systems, LNBIP 21, 2009, pp. 193–204.
[68] A.D. Birrell, B.J. Nelson, Implementing remote procedure calls, ACM Transactions on Computer Systems (TOCS) 2 (1) (1984) 39–59.
[69] A.S. Tanenbaum, R. van Renesse, A critique of the remote procedure call paradigm, in: Proceedings of the EUTECO 88 Conference, Elsevier Science Publishers, 1988, pp. 775–783.
[70] A. Barros, M. Dumas, A. ter Hofstede, Service interaction patterns, in: Proceedings 3rd International Conference on Business Process Management (BPM), Springer-Verlag, 2005, pp. 302–318.
[71] D.A. Chappell, Enterprise Service Bus, Theory in Practice, O'Reilly Media, 2004.
[72] F. Leymann, The (Service) bus: services penetrate everyday life, in: Proceedings of the 3rd International Conference on Service Oriented Computing (ICSOC), Springer-Verlag, 2005, pp. 382–386.
[73] H. Overdick, Towards resource-oriented BPEL, in: 2nd ECOWS Workshop on Emerging Web Services Technology, 2003, pp. 114–119.
[74] C. Pautasso, BPEL for REST, in: Proceedings of the 6th International Conference on Business Process Management, 2008, pp. 278–293.
[75] C. Pautasso, RESTful web service composition with BPEL for REST, Data & Knowledge Engineering 68 (9) (2009) 851–866.
[76] Apache ODE Group, BPEL Extensions, 2009. <http://ode.apache.org/bpel-extensions.html>.
[77] A. D'Ambrogio, P. Bocciarelli, A model-driven approach to describe and predict the performance of composite services, in: Proceedings of the 6th International Workshop on Software and Performance, 2007, pp. 78–89.

[78] W.T. Tsai, R. Paul, Y. Wang, C. Fan, D. Wang, Extending WSDL to facilitate web services testing, in: Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering, 2002, pp. 171–177.
[79] J. Farrell, H. Lausen, Semantic Annotations for WSDL and XML Schema, W3C Recommendation, 2007. <http://www.w3.org/TR/sawsdl/>.
[80] J. Recker, J. Mendling, On the translation between BPMN and BPEL: conceptual mismatch between process modeling languages, in: CAiSE 2006 Workshop Proceedings – Eleventh International Workshop on Exploring Modeling Methods in Systems Analysis and Design, 2006, pp. 521–532.
[81] A. Barros, G. Decker, A. Grosskopf, Complex events in business processes, in: Proceedings 10th International Conference on Business Information Systems, LNCS, vol. 4439, 2007, pp. 29–40.
[82] G. Decker, A. Grosskopf, A. Barros, A graphical notation for modeling complex events in business processes, in: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference, 2007, pp. 27–36.
[83] TM Forum, Business Process Framework (eTOM), 2010. <http://www.tmforum.org/BusinessProcessFramework/1647/home.html>.
[84] TM Forum, Solution Frameworks (NGOSS), 2010. <http://www.tmforum.org/SolutionFrameworks/1911/home.html>.
[85] TM Forum, Application Framework (TAM), 2010. <http://www.tmforum.org/ApplicationFramework/2322/home.html>.
[86] TM Forum, Information Framework (SID), 2010. <http://www.tmforum.org/InformationFramework/1684/home.html>.

Matjaz B. Juric, Ph.D., is a professor at the University of Maribor and the head of SOA Competency Centre. He has authored several SOA books, such as Business Process Driven SOA, SOA Approach to Integration, Business Process Execution Language, BPEL Cookbook (award for best SOA book in 2007), etc. Matjaz has been SOA consultant for several large companies. He has contributed to SOA Maturity Model and performance optimization of RMI-IIOP, etc. He is also a member of the BPEL Advisory Board.