Data-Bound Variables for WS-BPEL Executable Processes

Marcel Krizevnik*, Matjaz B. Juric
University of Ljubljana, Faculty of Computer and Information Science, Laboratory for Integration of Information Systems, Trzaska 25, SI-1000 Ljubljana, Slovenia

Abstract

Standard BPEL (Business Process Execution Language) variables, if used to store the data from a data store, cannot be automatically synchronized with the data source in case other applications change the data during the BPEL process execution, which is a common occurrence particularly for long-running BPEL processes. BPEL also does not provide a mechanism for active monitoring of changes of data that would support automated detection and handling of such changes. This paper proposes a new type of BPEL variables, called data-bound variables. Data-bound variables are automatically synchronized with the data source and thus eliminate the need to implement data synchronization manually. To provide support for data-bound variables, we propose specific extensions to BPEL and the use of appropriate DAS (Data Access Services) that act as data providers. We introduce new BPEL activities to load, create and delete remote data. We also introduce observed properties, observed property groups and a variable handler. Using this mechanism, the BPEL process is able to automatically adapt to changes to data, made inside or outside the process scope, by following the ECA (Event, Condition, Action) paradigm. As a proof-of-concept, we have developed a prototype implementation of our proposed BPEL extensions and tested it by implementing three pilot projects. We have confirmed that our proposed solution decreases BPEL process size and complexity, increases readability and reduces semantic gap between BPMN process model and BPEL.

Keywords: WS-BPEL, Service oriented architecture, Service composition, Data synchronization

1. Introduction

In this article, we address the problem of using out-of-date data in WS-BPEL 2.0 (Web Services Business Process Execution Language 2.0, or simply BPEL) processes. BPEL is a commonly accepted standard for defining business processes with composition of services in service oriented architecture (SOA) [19]. Standard BPEL variables are a snapshot of data returned by a service and thus present a duplicate version of remote data at some particular time. However, in certain cases we want to ensure that the process always uses the latest version of important business data as other applications may change the data in the data source during the BPEL process execution. This problem arises particularly in case of long-running BPEL processes, which can take a few days, weeks or even months to complete.

BPEL specification [17] does not provide any support for automated synchronization of variables. If we want to ensure that the BPEL process always uses the latest version of data, we have to implement data synchronization manually. This is usually done by adding additional service invocations as part of the business process flow. BPEL also does not provide a mechanism for active monitoring of variables that would allow us to define if-then rules that trigger corresponding actions in case the data has changed and the specified conditions are satisfied. BPMN (Business Process Model and Notation) 2.0 [8], on the other hand, supports the definition of such rules through the conditional event construct (rule event in BPMN 1.0) [5]. To implement similar functionality in BPEL, we have to perform manual checking in form of additional process steps. Adding such data synchronization and data checking code in several places in the business process can lead to a bad process design with code duplication and the code size and complexity are increased. As the complexity directly impacts readability [29,30], the readability of the business process is also reduced.

* Corresponding author. Tel.: +38641319451; fax: +38614264647.
E-mail addresses: marcel.krizevnik@fri.uni-lj.si, matjaz.juric@fri.uni-lj.si

To overcome these problems, we propose a new type of BPEL variables, called data-bound variables. Data-bound variables use SDO (Service Data Objects)-based form and are automatically synchronized with the data in the data source. They also support automated detection and handling of changes to data that were made inside or outside the BPEL process scope by following the ECA (Event, Condition, Action) paradigm [18], which is already supported in BPMN. In this way, we are able to remove most of the data synchronization and data checking code from the basic business process flow. Thus, we improve the BPEL process design. As we introduce support for a concept that is already supported in BPMN (data-based ECA rules), we also reduce semantic gap with BPMN and thus simplify the translation between BPMN and BPEL [11]. To provide support for data-bound variables, our solution extends BPEL and leverages the use of DAS (Data Access Services) that act as a data provider. We have designed the extensions for BPEL version 2.0 using the standard language extension mechanism.

The article is organized in nine sections and one appendix. In Section 2, we present motivation for providing automated synchronization between BPEL variables and the data source. Section 3 gives a brief overview of BPEL and SDO. In Section 4, we describe our proposed solution for data-bound variables from a high-level perspective. In Section 5, we describe the proposed solution for DAS and our extensions to the BPEL partner link type (BPEL extensions to WSDL) that are used to standardize the communication between the BPEL engine and the DAS. Section 6 describes the proposed extensions to BPEL, such as new activities and extensions to existing activities. In Section 7, we present the proof-of-concept, where we describe a procurement process case study and evaluate positive effects of using our proposed solution using business process metrics. In Section 8, we present related work and discuss the results. In Section 9, we give conclusions. Appendix defines syntax specification for proposed BPEL extensions.

2. Motivation

BPEL is widely used in different application domains, where special requirements and new challenges arise. One such important requirement is that BPEL process should always use the latest version of important business data. We present two types of business processes, where this requirement occurs:

- Long-running BPEL processes that have to follow strict laws and regulations. A typical example of such process is a procurement process in a large state-owned organization (for a complete motivating example please refer to Section 7), where some process instances can take a few weeks or even months to complete and some important business data (such as suppliers contact information, credit rating, list of references, status, etc) has to be refreshed at various steps in the process. Other examples of such long-running processes are loan approval process, inventory control process, etc.
- BPEL processes that operate with important data that changes very often. For example, a business process for booking airline tickets should always use the latest air fare prices, as these can change very often.

BPEL currently supports three types of variables: WSDL message type, XML simple type and XML schema element [19]. WSDL message type variables are the most commonly used type of variables and are used to store the data that is exchanged between business partners. Other two types of variables hold data which is used in business logic and for composing messages sent to partners. BPEL process variables may contain important business data (sometimes called master data [1]) that presents the core of the business process. If the organization has already performed data integration, master data may reside inside a central MDM (Master Data Management) repository [1]. Master data may include information about (but not limited to) customers, products, employees, materials, suppliers, etc. If such key business data is stored using a standard BPEL variable, the variable presents a duplicate version of remote data at some particular time. This means that the process may use out-of-date version of the data, as other applications may change the data during the BPEL process execution. In certain cases this does not present a problem, however, in some applications, the use of out-of-date data may result in invalid execution of key business activities [1,33]. Therefore, this problem has to be addressed. If, for example, a flight ticket booking process operates with out-of-date flight ticket data, it may not be able to find the cheapest ticket or it may try to book a ticket that is no longer available. Similarly, if the long-running procurement process does not continuously synchronize the list of suppliers, it may not detect that certain suppliers that have sent offers may have gone out of business or have been suspended during the process execution. For the similar reason, the loan approval process should always use customer's latest credit rating.

As the BPEL specification [17] does not provide a mechanism for automated synchronization of data, the BPEL developer has to implement these steps manually. This can be done by adding data synchronization steps into the process flow. To perform one such data synchronization, we need to add at least one `<invoke>` and two `<assign>` activities and introduce two variables that represent input and output for the service call. Furthermore, sometimes it is not enough just to ensure data synchronization, but we may also want to be able to automatically detect changes made by other applications so that the process is able to proactively react on these changes. For example, if CRM (Customer Relationship Management) system changes customer status to `INACTIVE`, we may want to immediately detect and appropriately handle this change in all active BPEL process instances using this customer. BPMN supports the definition of such ECA rules through the conditional event construct (rule event in BPMN 1.0) [5]. For example, we can use the event sub-process with conditional start event and thus isolate data checking and manipulation code from the main process flow, which improves the readability of the process. Such active monitoring of data is analogous to the data-based task trigger pattern [5]. To implement similar functionality in BPEL, the developer has to add `<if>` activities (to define the condition) and corresponding handler activities (to define the action that has to execute in case the condition evaluates to true). This approach has several drawbacks, as adding multiple data synchronization steps into the process flow results in code duplication and "pollutes" the flow [4]. Such approach may result in:

- Increased BPEL code size.
- Increased BPEL code complexity.
- Decreased readability of the BPEL code, as increased code complexity directly impacts readability [29,30].
- Difficult development and code maintenance, as the BPEL developer has to know at which steps to perform data synchronization and these steps are sometimes not directly reflected in the business process model. Code duplication also severely complicates the maintenance and evolution of enterprise solutions [2].

To solve the abovementioned problems, we decided to extend BPEL so that BPEL processes become data-aware. We introduce data-bound variables that are automatically synchronized with the data in the data source. We also introduce a mechanism called variable handler, which enables us to implement active monitoring of variables by following the ECA paradigm. Every change to key business data is automatically propagated to all active data-aware process instances using this data, so that the processes are able to adapt to the change. In some cases, they may only refresh their variables, while in others their behaviour may also be changed. The ability to monitor and adapt to the changes in external environment is very important in modern dynamic business environment [34]. As data-aware BPEL processes are capable of reacting to such changes, we also consider them to be environment-aware [32]. Added value of our proposed solution can be described as follows:

- BPEL process always uses the latest version of key business data.
- BPEL developer does not have to implement data synchronization manually.
- Data synchronization code is not duplicated at several locations in the process.
- Data changes, made within the business process instance, are automatically applied back to the data source.
- As the business process does not contain data synchronization code, BPEL code size and complexity are reduced and the code readability is improved. Our measurement results (please refer to Section 7) show that the overall size and complexity of the projects under test has been reduced by 6,4 % on average.
- The proposed variable handler enables us to define ECA rules for active monitoring of data. Thus, the BPEL process is able to automatically react on important changes to data (made

within the process or by other applications) that should change the behaviour of the business process.

- By introducing support for a concept that has been already supported in BPMN (data-based ECA rules), we also reduce semantic gap between BPMN and BPEL and ease the BPMN-BPEL round-tripping [11].

3. Brief overview of BPEL and SDO in aspect of proposed extensions

BPEL is a commonly accepted OASIS standard [17] that presents the cornerstone of SOA [19]. It supports the composition, orchestration and coordination of web services and presents a basis for the development of modern composite applications. With its ability to define executable and abstract business processes it plays an important role in business process management (BPM). The current version is 2.0, which has been approved by OASIS in April 2007. BPEL processes are defined as a collection of activities through which services are invoked. BPEL supports basic and structured activities. Basic activities present steps in the business process flow. We use `<invoke>` to invoke other web services. `<receive>` is used to wait for the client to invoke the business process by sending a message. `<reply>` is used for sending a response in synchronous operations. `<assign>` activity allows us to manipulate data variables. We use `<throw>` activity to signal internal faults. Using the `<rethrow>` activity we are able to rethrow faults that have been caught in fault handlers. `<exit>` can be used to immediately end the business process instance. `<wait>` allows us to wait for some time. `<empty>` activity does nothing and is most commonly used when a fault needs to be caught and suppressed or to provide a synchronization point in a `<flow>`. `<extensionActivity>` is part of BPEL extension mechanism and presents a wrapper for extension activities that are not defined by BPEL specification. We use the BPEL extension mechanism to add support for data-bound variables.

Basic activities can be combined using structured activities. `<sequence>` is used to define the order in which activities have to execute. We use `<if>` to implement conditional behaviour that follows the if-then pattern. `<while>` activity enables repeated execution of a contained activity, as long as the defined condition evaluates to true. `<repeatUntil>` activity is also used for repeated execution, however, the contained activity is executed until the defined condition becomes true. In contrast to `<while>`, `<repeatUntil>` loop executes at least once. We use `<pick>` when we want to wait for the occurrence of exactly one event from a set of events and then execute the activity that is associated with that event. Two types of events are supported: message events and timer events. `<flow>` activity is used for defining a set of activities that have to be invoked in parallel. `<forEach>` activity executes its contained activity N+1 times, where N is a counter value that can be set using an expression. Contained activity can be executed concurrently (parallel `<forEach>`) or in a sequence (serial `<forEach>`).

In addition to activities, BPEL also support scopes (`<scope>`) which provide a way to divide complex business processes into hierarchical organized parts. Scopes can be used to declare variables and to define different types of handlers. The most commonly used types of handlers are fault handlers that can be used to catch and handle faults.

SDO [20,21] is a data programming architecture and an API (Application Programming Interface) for data application development. One of the main goals when developing SDO was to simplify the data programming. SDO provides a uniform access to data from heterogeneous sources, so that developers can focus on the business logic instead of the underlying technology. With SDO, developers do not need to be familiar with technology specific APIs (such as JDBC, JAXB, DOM, SAX, JDO, JCA, JPA) to access the data. SDO architecture is based on the concept of disconnected data graphs. Basic concepts are data object, data graph and DAS (Data Access Service). Data object is a set of named properties, each of which represents a simple data value or a reference to another data object. Data graph presents the envelope around the data objects and is the basic unit for data transfer between different components. Every data graph may have a single root data object which contains all the other data objects in the data graph so that they form a tree or a graph. The root data object in the data graph represents the entity we are working with (for example customer). SDO metadata may be derived from XML schema, EMOF (Essential Meta Object Facility), Java, relational databases, or other structured representations. Data graphs can also store changes on data objects. These changes are written to the change summary, which is also part of the data graph. Access to data sources is provided by a set of components called DAS. The DAS is responsible for accessing the data source, creating data graphs and transmitting changes back to the data source.

We use SDO in our proposed extensions for data-bound variables, as SDO is currently the most appropriate data programming solution for SOA, as identified by several authors [31,33,37,38]. SDO is the only data programming technology that combines support for heterogeneous data source types, disconnected programming model [22], change tracking feature and support for static and dynamic data API. It is also very important that XML schema can be used to define the SDO model and that a subset of XPath 1.0 can be used for traversing through data objects [20]. If properly extended, the BPEL engine can easily manipulate SDO-based variables in a similar way as regular BPEL variables. For the above reasons, we claim that SDO is the most appropriate solution for our BPEL extensions for data-bound variables.

4. Overview of proposed solution for data-bound variables

The high-level overview of our proposed solution is shown in Fig. 1. There are three main components in the architecture: the BPEL engine, DAS and the data source. To provide support for data-bound variables, we have extended BPEL by introducing new BPEL activities and a variable handler. Every data-bound variable has to be bound to a corresponding DAS which provides access to data in the data source. The BPEL engine is responsible for performing automated synchronization between the local copy of data stored in data-bound variables and the data source by invoking DAS operations behind the scenes. The data transfer between the BPEL process and the DAS goes in form of SDO data graphs. The DAS has to provide basic CRUD (create, read, update, delete) data operations and the partner link type (`<partnerLinkType>`). We have proposed several extensions to the partner link type to define the metadata that enables automated communication between the BPEL engine and the DAS. Depending on the data source type, DAS uses the appropriate data programming technology (such as JDBC, ORM, or JAXB) to access the data.

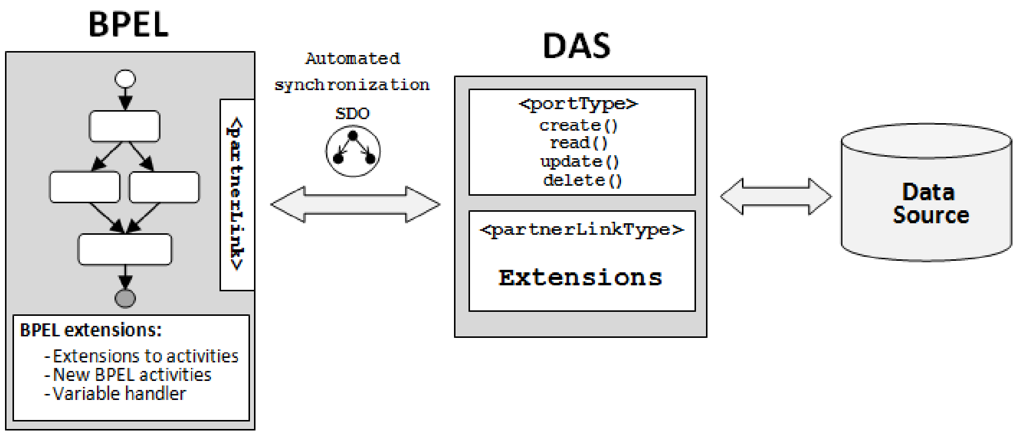

Fig. 1. High-level overview of the proposed solution

Data-bound variables use SDO-based in-memory data format to hold the data instead of DOM (Document Object Model)-based data format that is usually used. Each data-bound variable can hold one entity. Entity represents a "thing" which can be uniquely identified [23]. A specific customer order, offer, or stock can be an example of an entity. Entity can be presented as a simple or a complex SDO data object. The data object is simple if there is only a root data object that contains all the data. If the root data object contains references to other data objects (for example customer order can contain multiple order items), we talk about complex data object.

Data-bound variables are not persisted to the BPEL dehydration store as part of the process instance state. BPEL dehydration [19] is a process of storing the process state into the database (usually referred to as a dehydration store) to free up engine resources. This happens particularly in asynchronous scenarios where BPEL process invokes a partner link service (using the `<invoke>` activity) and then waits for the response (using the `<receive>` or `<pick>` activities). When the engine receives the response, it restores the process with its state from the database (hydration) and then continues with the execution of the process. For efficient execution of long-running business processes the BPEL engine has to support dehydration, therefore it is supported by all available BPEL engines. In our proposed solution, BPEL hydration/dehydration points are used for performing synchronization between the local copy and remote data. Before every process instance dehydration, changes to data in the data-bound variables (if any) are applied to the data source. Similarly, as part of the process instance hydration, fresh data from remote data source is loaded into the data-bound variables. To implement proposed extensions, the BPEL engine has to implement SDO specification, to be able to work with SDOs. In some existing BPEL engines (such as IBM WebSphere Process Server [43]) all standard BPEL variables are already represented as SDOs. The BPEL engine has to be able to automatically convert between the regular (in most cases DOM-based) BPEL variables and the data-bound variables. For example, we should be able to copy data from regular BPEL variable to data-bound variable and vice-versa using a standard `<assign>` activity. When accessing data stored in data-bound variables, the user always refers to the latest version of data. The user cannot refer to the value of the variable before it has changed.

Every data-bound variable must have a key (id), which uniquely identifies stored entity and points to the specific record in the data source. In case of a relational database, this is the primary key of the table. Only the key of the data-bound variable is stored to the BPEL dehydration store. The key is a read-only attribute and should not be changed from BPEL process using the `<assign>` activity. Every attempt to change the key should result in an exception. To be able to use data-bound variable, we need to implement a corresponding DAS, which is used to populate the variable from the data source. DAS is also responsible for applying changes to the variable in the opposite direction. The DAS architecture depends on the back-end data source type. For example, XML DAS loads and saves the data graph as an XML file and a JDBC DAS loads and saves the data graph using a relational database. Our proposed solution requires DAS to be a service that provides appropriate CRUD operations to work with the data, regardless of the back-end data source type. For every entity type that we want to use, we have to develop a corresponding DAS. For example, to be able to store customer order, we have to implement a DAS that provides operations for creating, reading, updating and deleting customer orders. However, every DAS can be reused by other BPEL processes and also by any other type of client that supports SDO and is capable of invoking services.

Fig. 2 shows a simple scenario, where we demonstrate how the data-bound variable can be used to perform automated data synchronization. However, our proposed solution also provides other important functionalities, such as active monitoring of data by using variables handlers. Those advanced concepts shall be presented in details in Section 6.

The first step is to load an existing entity from the data source or to create a new entity. Once the variable is bound to the entity in the data source, we do not need to add any more BPEL code to perform synchronization. To be able to load an entity, we have to set its key. This can be done using the `<bpelx:loadEntity>` BPEL extension activity, which we describe in detail in Section 6. The result of loading an entity at run time is invocation of DAS `read()` operation. DAS reads the data from the persistence store and builds the data graph. To be able to return the data graph to BPEL, the data graph has to be serialized as an XML stream. XML format for data graph serialization is defined in the SDO specification [20]. After receiving the data graph, the BPEL engine has to de-serialize it. Then, it is able to populate the data-bound variable with the most recent data and place it into working memory. The BPEL process is then able to use the variable in the same way as any regular BPEL variable. We can also change the data using a standard `<assign>` activity. All changes to the local copy of the data are tracked using the SDO change summary feature. Just before next BPEL dehydration (and at the end of the process), the BPEL engine automatically propagates the modified data graph (which also includes the change summary) to DAS by invoking the `update()` operation. If the data has not been changed, this step is skipped. When DAS receives the data graph that contains the modified data objects and the change summary, it writes the changes to the persistence store. Then, the SDO data objects can be removed from the working memory and the BPEL process dehydrates.

While the BPEL process instance is dehydrated, only the data-bound variable key is stored to the BPEL dehydration store. If other applications change the data in the data source while the BPEL process is dehydrated, these changes also have to be propagated to the BPEL process instance. Therefore, during BPEL process hydration, the BPEL engine automatically invokes DAS `read()`

operation and passes the data-bound variable key to retrieve the latest version of the data. Following the described approach, we can ensure that BPEL process always works with the latest version of important business data.

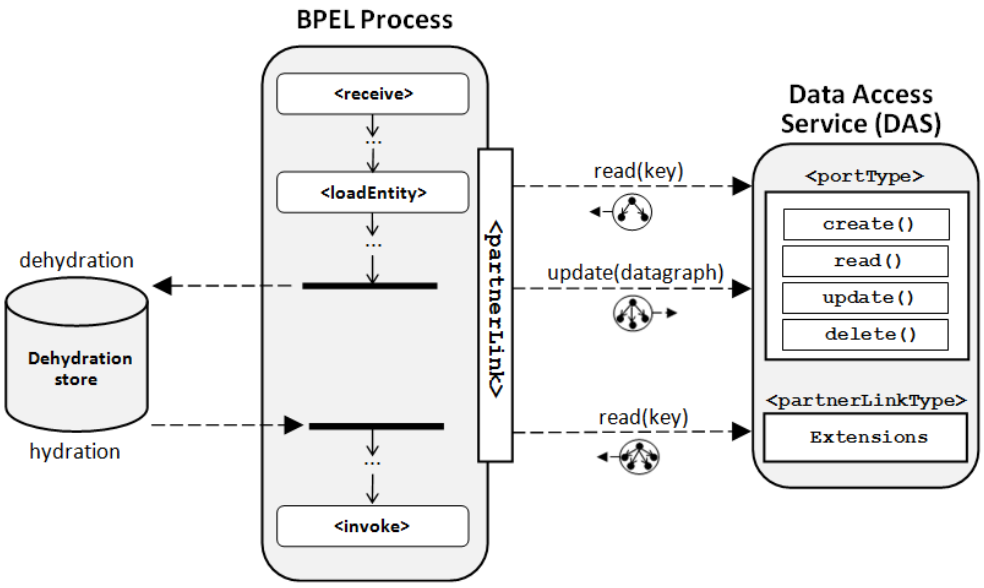

Fig. 2. Typical scenario of using a data-bound variable in BPEL

By default, changes to data-bound variables are applied to the data source only before every BPEL dehydration and at the end of the process and fresh data is only loaded as part of process hydration. However, we can also increase the data synchronization frequency by configuring the variable synchronization mode during variable declaration (please refer to Section 6.1). In this way, we can achieve that every change to data-bound variable is immediately propagated to the data source and that fresh data is loaded every time we read the data from the data-bound variable. However, the drawback of this type of synchronization can be decreased performance, due to increased number of DAS invocations.

The implementation of the DAS depends on the specific situation and on the data source type. Our proposed solution follows the disconnected data programming model. DAS typically uses optimistic concurrency control (OCC) to avoid locking. In case of concurrent updates of the same data, DAS may require to know which properties have been changed and what the previous values were to detect update collisions. The change summary in the data graph provides full history of the changes. Before committing, the collision detection strategy compares current values in the data source to the old values from the change summary. If the values match, the update operation is performed, otherwise an exception can be thrown or other conflict resolution strategies can be used. For example, DAS can try to merge the changes or even ignore the collision and overwrite the data. In case DAS update operation fails, an exception can be handled in business processes code using standard BPEL fault handling mechanism. OCC provides better performance and is particularly suitable for disconnected environments where the possibility of conflicts is low. However, if there are lot of updates and if conflicts are very likely, DAS can use pessimistic concurrency control (PCC). PCC (also called pessimistic locking) is an approach that assumes that two or more users will try to update the same entity at the same time and prevents the possibility of update collisions by locking entities in the data source. The advantage of PCC is that it is easier to implement and that it guarantees that all changes to the data source are made consistently and safely. However, this approach is not scalable and when a system has many users, or when the transactions are long-lived, the chance of having to wait for a lock to be released increases which can impact performance. Therefore, PCC has to be used with caution and an appropriate lock timeout has to be set. If the update operation fails (e.g., in case of a lock timeout), DAS can throw an exception and the BPEL engine may retry with invocation within a specified retry interval, which can be set as a BPEL engine property.

Although the data transfer between the BPEL process and the DAS goes in form of data graph envelope object, which can also include SDO model and the change summary, BPEL developers do not need to work with data graphs directly. They just work with a tree of data objects that represent the entity in the same way as with regular BPEL variables. The BPEL engine is responsible for holding the data graph and for tracking the changes to data objects by generating the change summary.

In the following two sections, we describe our proposed solution in detail. In Section 5, we describe proposed solution for DAS. We describe data graph serialization, CRUD operations and the extensions to the partner link type. In Section 6, we present proposed extensions to BPEL, which includes new BPEL activities and a variable handler.

5. Proposed solution for DAS

To enable automated communication between the BPEL engine and the DAS, the DAS interface has to be properly defined. It has to expose basic CRUD data operations. In Section 5.1, we describe these operations. In Section 5.2, we present extensions to the BPEL partner link type which has to clearly describe DAS interface by defining metadata (pointers to CRUD operations, data graph, root data object and key) that enables the BPEL engine to automatically invoke DAS operations.

5.1. Proposed DAS interface

DAS has to provide the following operations:

- `createEntity(): dataGraph`. When creating a new entity, DAS first has to retrieve a key that uniquely identifies the new entity. However, DAS does not yet insert the data into the data source. Instead, it builds new data graph with empty root data object and only sets its key. This data graph is then returned to the client (BPEL engine). The BPEL process can then set the data objects. To actually insert new data to the data source, the BPEL engine also has to invoke the `updateEntity()` operation. This is done automatically just before next dehydration or at the end of the process.
- `readEntity(key): dataGraph`. The operation receives the key which uniquely identifies the entity. DAS retrieves the data from the data source, builds the data graph, serializes it and returns it to the BPEL engine.
- `updateEntity(dataGraph)`. The operation accepts the modified data graph, which also includes the change summary. DAS has to de-serialize the data graph and write the changes to the data source. However, before writing the changes, DAS has to check whether the entity already exists in the data source or not (in case we just created it by invoking the `createEntity()` operation). If the entity already exists, DAS updates the data, otherwise it inserts it.
- `deleteEntity(key)`. This operation accepts the key of the entity we want to delete. When deleting, DAS has to follow constraints and relationships between objects, as defined in the SDO model.

`Entity` in operation names presents the name of the entity we are working with. Actual method names may be `createCustomer()`, `readCustomer()`, `updateCustomer()` and `deleteCustomer()`. However, as we provide the mapping for those operations in our partner link type extensions, operation names may also follow different naming convention. In order to develop a reliable and cost-effective software, faults have to be appropriately detected and handled. Therefore, it is strongly recommended to define a fault for every DAS operation, which triggers in case the operation fails. This enables us to catch and handle those faults in BPEL process using a standard BPEL fault handling mechanism.

5.2. Proposed extensions to the BPEL partner link type

To be able to bind data-bound variables to DAS and to enable the BPEL engine to automatically invoke DAS operations, DAS WSDL has to clearly describe its interface by specifying the following metadata:

- Mapping for CRUD operations
- Reference to the data graph
- Reference to the root data object that represents the entity
- Reference to the element that defines the entity key

We decided to place this metadata as an extension to the BPEL partner link type (`<partnerLinkType>`) construct, as the partner link type defines the nature of the interaction between the BPEL process and the partner service. The partner link type is a BPEL extension to WSDL [13,14] and is part of the BPEL specification [17]. Each partner link type defines one (synchronous scenario) or two (asynchronous scenario) roles and lists the port types that each role must support for the interaction to be carried out successfully. The partner link type can be used to represent dependencies between services, whether or not a BPEL process is defined for one or more of those services. That is the reason why the partner link type is defined in a WSDL file and not in a BPEL process file.

To enable the DAS to act as a data provider for the BPEL data-bound variable, we have added the `<plnkx:dataAccessServiceInstance>` construct to the `<partnerLinkType>` section of the WSDL, as shown in example in Listing 1. We use the XML namespace extension mechanism, which allows us to introduce extensions while assuring compatibility with other service clients that are unaware of these extensions. To achieve this, we have introduced a namespace URI `http://soa.si/das/databoundvar`, for which we use the alias `plnkx`. The formal syntax specification for the presented partner link type extension is defined in Appendix.

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" ...>
  ...
  <wsdl:message name="ReadCustomerRequest">
    <wsdl:part name="body" element="cus:customerKey"/>
  </wsdl:message>
  <wsdl:message name="ReadCustomerResponse">
    <wsdl:part name="body" element="cus:customerDataGraph"/>
  </wsdl:message>
  ...
  <!-- Declaration of a BPEL property that represents a key part -->
  <vprop:property name="customerId" type="xs:int"/>
  <!-- Mappings for key parts -->
  <vprop:propertyAlias propertyName="tns:customerId" element="cus:customer">
    <bpel:query>cus:customerId</bpel:query>
  </vprop:propertyAlias>
  <vprop:propertyAlias propertyName="tns:customerId" element="cus:customerKey">
    <bpel:query>cus:customerId</bpel:query>
  </vprop:propertyAlias>

  <wsdl:portType name="CustomerPT">
     <wsdl:operation name="readCustomer">
       <wsdl:input message="tns:ReadCustomerRequest"/>
       <wsdl:output message="tns:ReadCustomerResponse"/>
       <wsdl:fault name="ReadCustomerFault" message="tns:FaultMessage"/>
     </wsdl:operation>
     ...
  </wsdl:portType>
  <plnk:partnerLinkType name="CustomerPLT">
    <plnk:role name="CustomerDASProvider" portType="tns:CustomerPT"/>

    <!-- Metadata that enables automated communication between BPEL
         engine and DAS -->
    <plnkx:dataAccessServiceInstance name="CustomerService"
           dataGraph="cus:customerDataGraph"
           entity="cus:customer"
           key="cus:customerKey"
           keyParts="tns:customerId"
           createOpName="createCustomer"
           readOpName="readCustomer"
           updateOpName="updateCustomer"
```

```
          deleteOpName="deleteCustomer"/>
  </plnk:partnerLinkType>
  ...
<wsdl:definitions>
```

Listing 1. Proposed extensions to the BPEL partner link type

The `<plnkx:dataAccessServiceInstance>` construct defines all information the BPEL engine needs to be able to automatically invoke DAS. The `dataGraph` attribute defines the data graph. The `entity` attribute references the root data object that represents the entity type we are working with. As every root data object needs a unique key and as the key may be compound, the `keyParts` attribute specifies a list of all the key parts using the space separator. Every key part is defined using a BPEL `<property>`. BPEL properties [17] are BPEL extension to WSDL which allow us to associate relevant data with names that have greater significance than just data types used for such data. BPEL properties are used to define different kinds of unique identifiers which appear in different messages and can also be used for correlation. To map a property to a specific element, BPEL provides property aliases (`<propertyAlias>`). With property aliases, we map a property to a specific element of a message part or another element.

For every key part, we define one property and set its name and type. In example in Listing 1, the key has one part (`customerId`), therefore we define only one `<property>` element. Every key part is defined in two elements: in the root data object (`customer`) and in the element that represents the key (`customerKey`). This element presents an argument for `read()` and `delete()` operations and is explicitly defined using the `key` attribute. For the BPEL engine to be able to automatically correlate between key parts in both elements, we have to define two `<propertyAlias>` mappings for each key part. In case of a composite key, the `keyParts` attribute defines all the key parts, so that key part property names are listed using the space separator. The `createOpName`, `readOpName`, `updateOpName` and `deleteOpName` attributes provide the mappings for CRUD data operations and thus allow the DAS to use different method naming conventions.

6. Proposed extensions to BPEL

In this Section, we propose specific extensions to BPEL to add support for data-bound variables. The proposed extensions use the standard BPEL extension mechanism `<extensionActivity>` and can therefore be implemented on any BPEL 2.0 compliant engine. In accordance with specification, all extension activities support standard attributes and elements. For extensions we use the URI `http://soa.si/bpel/databoundvar` namespace, for which we use the alias `bpelx`. Full syntax specification of the extensions is provided in Appendix.

6.1. Extensions to variable declaration

Every data-bound variable has to be bound to DAS. An example of a data-bound variable declaration is shown in Listing 2. To bind the variable to DAS, we use the `partnerLink` attribute. We can also set the data synchronization mode using the `bpelx:synchMode` attribute. The attribute is optional and if not present, the default value (`lazy`) is used. This means that changes to data-bound variables are applied to the data source only before every BPEL dehydration and at the end of the process. Similarly, fresh data from the data source is loaded only during every BPEL hydration. By setting the `bpelx:synchMode` attribute to value `eager`, we can ensure, that every change to data-bound variable is immediately propagated to the data source and that fresh data is loaded every time we read the data. Eager synchronization mode is appropriate for both short- and long-running business processes that operate with important data that changes very often. However, the drawback of eager synchronization is decreased performance.

Data-bound variables can be marked as read-only. This can be done using the `bpelx:readOnly` attribute. The attribute value can be set to `yes` or `no`. The use of `bpelx:readOnly` attribute is optional. If the attribute is not present, the default value (`no`) is used. When using read-only data-bound variables, we are not allowed to change variable from the BPEL process and DAS `update()` operation is not invoked before the BPEL dehydration. This increases robustness of the BPEL code, as it assures that BPEL developers cannot change the data by mistake and also potentially improves performance due to reduced number of DAS invocations.

```
<variable name="customer"
          partnerLink="CustomerService"
          bpelx:synchMode="lazy" bpelx:readOnly="no" />
```

Listing 2. Declaration of a data-bound variable

Similar to regular BPEL variables, data-bound variables are declared within a `<scope>` and are said to belong to that scope. Data-bound variables that belong to the global scope are called global data-bound variables. Data-bound variables may also belong to other, non-global scopes and such variables are called local data-bound variables.

During data-bound variable declaration, we can also specify so called observed properties and observed property groups, which are needed when using variable handlers. Observed properties and variable handlers shall be discussed in detail in Section 6.4.

Before data-bound variable can be used it has to be initialized. Data-bound variable can be initialized by defining which specific entity in the data source it represents. This can be done using the `<bpelx:loadEntity>` extension activity. Another option to start using the data-bound variable is to create a new entity, for which the `<bpelx:createEntity>` extension activity is used.

6.2. Loading entities

After the data-bound variable has been declared, we have to bind it to the data source by setting the entity key. To bind the variable and to load the data, we introduce the `<bpelx:loadEntity>` activity. Listing 3 shows an example of loading an entity. We set the activity name (`name` attribute) and the data-bound variable name (`variable` attribute). Using the `<bpelx:keyPart>` element we can set entity key by specifying an XPath expression. In case of a composite key, there has to be more `<bpelx:keyPart>` elements, each setting a part of the key. For each `<bpelx:keyPart>` element, we have to set a corresponding BPEL property (`property` attribute) that is defined in DAS partner link type. In example shown in Listing 3, the key has only one part and we set its value by assigning it a customer identifier from a BPEL variable called `$input`. The result of the `<bpelx:loadEntity>` activity at run time is invocation of DAS `read()` operation, where the key is passed as an argument.

```
<extensionActivity>
  <bpelx:loadEntity name="loadCustomer" variable="customer">
    <bpelx:keyPart property="ns1:customerId">
       $input.travelRequest/ns2:customerId
    </bpelx:keyPart>
  </bpelx:loadEntity>
</extensionActivity>
```

Listing 3. Loading a data-bound variable

6.3. Creating and deleting entities

In certain cases, we may also want to create or delete an entity from the BPEL process. Therefore, we introduce `<bpelx:createEntity>` and `<bpelx:deleteEntity>` activities. These activities cannot be used with read-only data-bound variables. The `<bpelx:createEntity>` activity is used for creating a new entity by invoking DAS `create()` operation. An example of using the `<bpelx:createEntity>` activity is shown in Listing 4. As we can see, the use of this activity is very simple as we only set the activity name and the data-bound variable name.

```
<extensionActivity>
  <bpelx:createEntity name="createCustomer" variable="customer" />
</extensionActivity>
```

Listing 4. Creating a new entity

Fig. 3 shows a typical scenario of creating a new entity. When DAS `create()` operation is invoked (1), DAS first retrieves a unique key from the data source (2). In case of a relational database, the key can be retrieved using a sequence. DAS then inserts an empty root data object into the data graph and returns serialized data graph to the BPEL. DAS does not yet insert the data into the data source. After retrieving the data graph with empty root data object (only the key is set), BPEL can set the data using the `<assign>` activity. Just before next BPEL dehydration, the BPEL engine automatically invokes DAS `update()`operation and passes the modified data objects and the change summary (3). DAS checks whether the entity with this key already exists in the data source or not. In this case, the entity does not yet exist, therefore the insert operation is performed (4). While the BPEL process instance is dehydrated, other applications may change the entity in the data source. During next BPEL hydration, the BPEL engine automatically retrieves fresh data by invoking DAS `read()`operation (5).

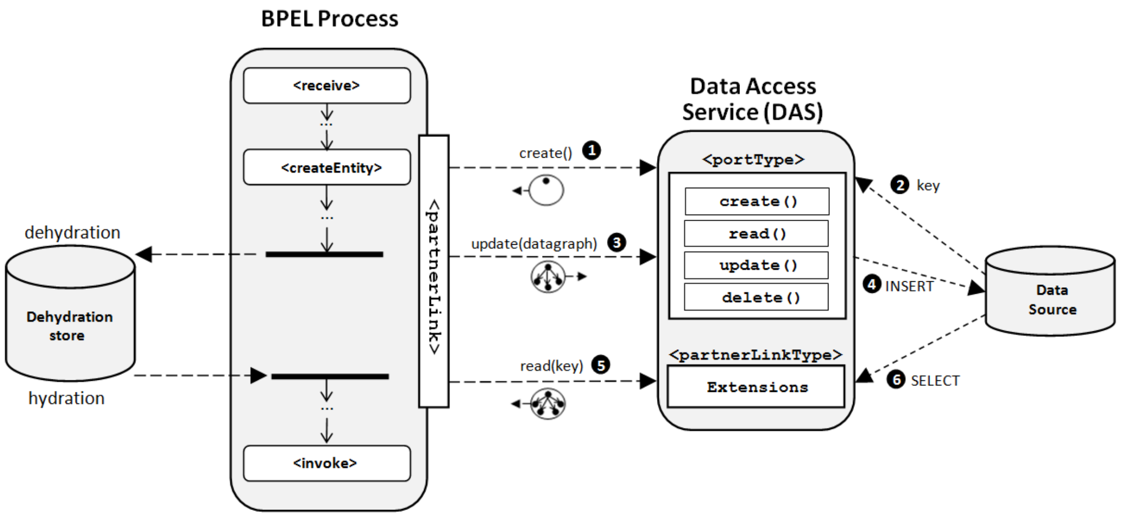

Fig. 3. Typical scenario of creating a new entity

The `<bpelx:deleteEntity>` activity allows us to delete remote entities. The activity can only be used after the data-bound variable has already been initialized. Any attempt to delete an entity using uninitialized variable should result in an exception. Listing 5 shows an example of using the activity. We only have to set the activity name and the data-bound variable name.

```
<extensionActivity>
  <bpelx:deleteEntity name="deleteCustomer" variable="customer" />
</extensionActivity>
```

Listing 5. Deleting an entity

When deleting, DAS has to follow constraints and relationships between objects, as defined in the SDO model. Typical scenario of deleting an entity is shown in Fig. 4. After the variable has been initialized (1,2), we are able to delete the entity using the `<bpelx:deleteEntity>` activity (3). Behind the scenes, DAS `delete()` operation is invoked and the entity is removed from the data source (4).

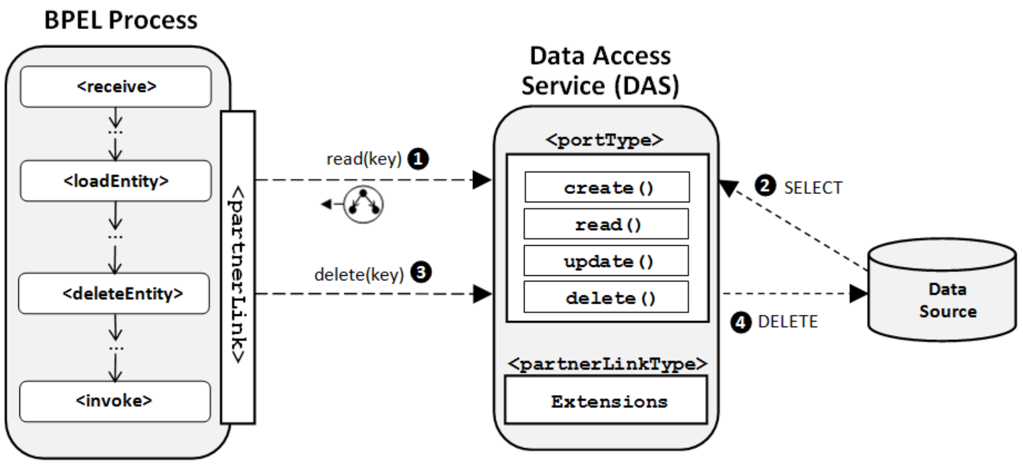

Fig. 4. Typical scenario of deleting an entity

6.4. Observed properties, observed property groups and variable handlers

In some cases, it is not enough just to ensure that the BPEL process automatically retrieves the most recent data. We may also need to know which data has been changed to be able to handle this in the BPEL code. To enable the detection and handling of changes to data stored in data-bound variables we introduce observed properties, observed property groups and a variable handler. Using variable handlers, we are able to define a set of if-then rules that are evaluated for every data change. This enables the BPEL process to detect and react on the changes, where these changes can be made by an external application or within the process flow. Thus, we follow the ECA (Event-Condition-Action) paradigm. The ECA rule has the general syntax [18,42]:

on `event` **if** `condition` **do** `actions`

In our solution, event presents a change of a data-bound variable. The condition is defined using the XPath expression inside the `<condition>` construct of the variable handler. The action part of the variable handler defines a sequence of BPEL activities that have to execute in case the condition is evaluated to true.

However, to be able to use variable handlers, we first have to define so called observed properties. Every observed property presents a field (sub-element) of a data-bound variable, where this field requires a special attention as the change of its value should change the normal flow of the BPEL process. Observed properties can only be defined for fields of a simple type and therefore present atomic values. A customer status is a good example for an observed property. We can define more observed properties for a data-bound variable. Only changes to observed properties can be monitored using the variable handler. Observed properties can be defined during data-bound variable declaration, using a new `<bpelx:observedProperty>` extension element. For each observed property we have to set the name (`name` attribute) and the XPath expression that points to the field, which the observed property is related to (`query` element). The XPath expression is always relative to the root data object which is defined in the DAS partner link type extensions. Observed properties for a particular data-bound variable are nested within the `<bpelx:observedProperties>` extension element, as shown in Listing 6. As the BPEL engine has to be able to detect changes to observed properties, it always has to store the values of all the observed properties before dehydrating a particular BPEL process instance. Therefore, when using observed properties, not only the key of the data-bound variable is stored to the BPEL dehydration store but also values of all the observed properties. During the BPEL hydration, these old values are compared to the values returned from DAS. If the BPEL engine detects the change, the process instance is able to handle it if a proper variable handler has been defined.

Observed properties can also be grouped using observed property groups that present multi-valued attributes. We use observed property groups when there are more logically-related observed properties and we want to be able to handle a situation when at least one of the properties defined within the property group has changed. Observed property groups are defined using a new `<bpelx:observedPropertyGroup>` extension element. For every observed property group, we have to specify observed properties that belong to that group by setting the `observedProperties` attribute, where observed property names have to be listed using the space delimiter. All observed property groups for a specific data-bound variable are defined within the `<bpelx:observedPropertyGroups>` extension element. Listing 6 demonstrates the definition of observed properties and observed property groups as part of data-bound variable declaration. We declare five observed properties. The first observed property presents customer status. The next four properties present customer address and together form an observed property group, named `address`.

```
<variable name="customer" partnerLink="CustomerService"
          bpelx:synchMode="lazy" bpelx:readOnly="no">

   <bpelx:observedProperties>
     <bpelx:observedProperty name="status">
       <query>cus:status</query>
     </bpelx:observedProperty>
     <bpelx:observedProperty name="number">
       <query>cus:address/cus:number</query>
     </bpelx:observedProperty>
     <bpelx:observedProperty name="street">
       <query>cus:address/cus:street</query>
     </bpelx:observedProperty>
     <bpelx:observedProperty name="city">
       <query>cus:address/cus:city</query>
     </bpelx:observedProperty>
     <bpelx:observedProperty name="country">
       <query>cus:address/cus:country</query>
     </bpelx:observedProperty>
   </bpelx:observedProperties>
   <bpelx:observedPropertyGroups>
     <bpelx:observedPropertyGroup name="address"
            observedProperties="number street city country"/>
   </bpelx:observedPropertyGroups>
</variable>
```

Listing 6. Declaration of observed properties and observed property groups

Variable handlers can only be used with data-bound variables and if appropriate observed properties have been defined. Variable handlers are defined within a new `<bpelx:variableHandlers>` section. We nest specific `<bpelx:onObservedPropertyChange>` and `<bpelx:onObservedPropertyGroupChange>` elements within the variable handlers section. The variable handler can be applied to a process or a scope, similar as fault or event handler. It defines the behaviour that is executed when a certain observed property or one or more observed properties within observed property group are changed. This enables us to gain control over data changes.

For every `<bpelx:onObservedPropertyChange>` element, we have to specify the variable name (`variable` attribute) and name of the observed property (`observedProperty` attribute) we want to handle. In some cases, it suffices to know whether observed property has been changed or not. However, we can also apply a filter and specify a condition using the `<condition>` construct. In that case, the variable handler will execute only if the observed property has changed and if the condition evaluates to true. Thus, we can define ECA rules for active data monitoring.

For every `<bpelx:onObservedPropertyGroupChange>` element, we have to set the data-bound variable name (`variable` attribute) and the observed property group name (`observedPropertyGroup` attribute).

Using the `type` attribute, variable handler can be declared either as interrupting or non-interrupting. If the `type` attribute is not specified, the default value (interrupting) is used. When interrupting variable handler triggers, it terminates the execution of the handled scope, similar as a fault handler. A non-interrupting variable handler, on the other hand, executes concurrently with the main flow. The selection of the appropriate variable handler type depends on the specific business process requirements. For example, a variable handler that executes in case stocks fall below defined minimum and contains logic for buying stocks, may be defined as non-interrupting. On the other hand, handler that triggers in case customer status has changed to `INACTIVE` should in most cases be defined as interrupting. The concept of interrupting and non-interrupting events is already known in BPMN 2.0 [8]. In general, when transforming BPMN process diagram to BPEL, BPMN event sub-processes should be translated into BPEL handlers [8]. For example, error event sub-process should be mapped to BPEL fault handler and compensation event sub-process should be mapped to a BPEL compensation handler. BPEL does not support a mechanism, similar to BPMN conditional event sub-process. Therefore, event sub-processes cannot be directly mapped. By introducing support for variable handlers, BPMN event sub-processes with conditional start event (both interrupting and non-interrupting) can be directly translated into BPEL variable handlers, and vice versa. In this way, we have reduced semantic gap between BPMN and BPEL.

Listing 7 demonstrates the use of variable handlers by defining two handlers. The first handler is applied to the entire process and executes only if the status of the customer is changed to value `INACTIVE`. The second variable handler is defined within a scope and triggers if at least one of the observed properties defined within the group `address` has changed.

```
<process name="purchaseOrderProcess"
         targetNamespace="http://soa.si/purchaseOrder/"
         xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
         xmlns:bpelx="http://soa.si/bpel/databoundvar" ...>
  ...
  <extensions>
   <extension namespace="http://soa.si/bpel/databoundvar"
              mustUnderstand="yes"/>
  </extensions>
  <import.../>
  <partnerLinks>...</partnerLinks>
  <messageExchanges>...</messageExchanges>
  <variables>
    <!-- Data-bound variable declaration, same as in Listing 6 -->
    ...
  </variables>
  <correlationSets>...</correlationSets>
  <faultHandlers>...</faultHandlers>
  <eventHandlers>...</eventHandlers>
  <bpelx:variableHandlers>
    <bpelx:onObservedPropertyChange variable="customer"
           observedProperty="status">
     <condition>$customer/cus:status = 'INACTIVE'</condition>
     <sequence>
       <!-- If custumer status is changed to INACTIVE, do the following -->
     </sequence>
    </bpelx:onObservedPropertyChange>
  </bpelx:variableHandlers>
  <sequence>
    <receive.../>
    <scope name="scope1">
      <variables>...</variables>
      ...
      <terminationHandler>...</terminationHandler>
      <bpelx:variableHandlers>
        <bpelx:onObservedPropertyGroupChange variable="customer"
               observedPropertyGroup="address" type="non-interrupting">
          <!-- If at least one of the observed properties within the group
               address is changed, this handler will execute -->
          <sequence>
          ...
          </sequence>
        </bpelx:onObservedPropertyGroupChange>
      </bpelx:variableHandlers>
      <sequence>
        <!-- Scope flow is defined here -->
        ...
      </sequence>
    </scope>
    ...
  <sequence>
</process>
```

Listing 7. Example of using variable handlers

7. Proof-of-concept

For the proof-of-concept we have developed a prototype implementation of our proposed BPEL extensions. As part of testing, we were able to verify the extensions by implementing three real-world pilot projects, which we had already developed in the past for various clients. In Section 7.1, we describe the implementation of the prototype. In Section 7.2, we present a procurement process case study. In Section 7.3, we present and analyze the results.

7.1. Implementation of proposed extensions

In general, there are two approaches for the realization of the extensions [46]. We can extend the BPEL engine (or even implement a new one), or we can develop a tool that performs the translation of the extended BPEL into standard BPEL. The second option seems much easier to implement and offers high reusability and portability, as the translated BPEL code can be executed on any BPEL-compliant engine, without the need to modify it. However, this approach is not applicable in all cases, as all extensions cannot be applied without modifying the BPEL runtime environment [46]. Another disadvantage of this approach is that it can be hard to perform monitoring and debugging, as the code that gets executed is not the same as the code with extensions. Extending the BPEL engine, on the other hand, provides more holistic and consistent integration of the extensions in the runtime environment and can provide better performance due to more optimized process code and reduced communication overhead. However, it requires a lot of development effort. In our proposed solution, data-bound variables are SDO-based, which means that BPEL runtime has to support SDOs. SDO provides some important functionality, such as support for disconnected data programming model, static and dynamic data API and the change tracking feature. Especially the change tracking feature is very important in our solution, as the change summary is used by DAS to detect update collisions when using the optimistic concurrency control, which is the preferred approach in the disconnected environments. By translating our extended BPEL into standard BPEL, data-bound variables would be treated as regular BPEL variables and change tracking would not be supported. Even in the BPEL engines that already support SDO-based variables, the change summary is not passed when invoking external services. For the above reasons, we have decided to implement our proposed solution by extending an open-source BPEL engine.

The prototype has been implemented as an extension of the ActiveBPEL engine [10] (version 5.0). The initial prototype implementation has taken nine person-months and has been done by a group of three senior developers. ActiveBPEL is a lightweight open-source BPEL runtime environment written in Java that provides full support for BPEL 2.0 and can run in every Java servlet container, such as Apache Tomcat. ActiveBPEL has been released by Active Endpoints under the GPL license, however, they have stopped maintaining the open-source engine and now focus only on their commercial product ActiveVOS.

Our proposed approach can be applied to any standards-compliant BPEL server. In order to implement the extensions, the BPEL engine has to be extended with the support for data-bound variables, which requires:

- The BPEL engine has to implement SDO specification to be able to work with SDOs. An existing open-source SDO implementation (such as EclipseLink SDO or Apache Tuscany SDO) can be used.

- The BPEL engine has to support variable declaration extensions, extension activities and the variable handler.
- To be able to automatically invoke DAS operations, the BPEL engine has to support the partner link type extensions that reside in DAS WSDL inside the `<plnkx:dataAccessServiceInstance>` construct.
- The BPEL engine has to activate the SDO change summary feature. This functionality is provided by SDO implementation and can be activated using the `getChangeSummary().beginLogging()` command.
- The BPEL hydration/dehydration process has to be extended. During every dehydration, DAS `update()` operation has to be automatically invoked and the entity key and all related observed properties (if they exist) have to be persisted to the dehydration store. During each hydration, fresh data has to be automatically retrieved by invoking DAS `read()` operation.
- After retrieving fresh data, the BPEL engine always has to check if the data has changed and trigger corresponding variable handlers if they exist and if specified conditions are satisfied.
- The BPEL engine has to be capable of copying data from regular BPEL variables to data-bound variables and vice-versa using standard `<assign>` activity. XPath 1.0 expression can be used to navigate through SDO data graph.
- The BPEL process auditing mechanism has to be extended to support auditing of the proposed extension activities and variables.
- The extended BPEL engine has to support validation of BPEL process source files with the data-bound variable extensions.

7.2. Case study: Procurement process

We will describe the usability of proposed extensions on a complex real-world procurement process for a large state-owned power distribution company. We received a BPMN process model as part of the specification. This model presented a basis for the BPEL implementation. Due to the business process complexity, the BPMN process model is structured using sub-processes, so that the total number of BPMN processes is 12. For each BPMN (sub)process we implemented a corresponding BPEL process. The solution consists of totally 12 BPEL processes that compose four level process hierarchy. Fig. 5 shows the top-level BPMN process diagram that orchestrates the high-level process steps: order preparation, offer collection, offer selection and order completion.

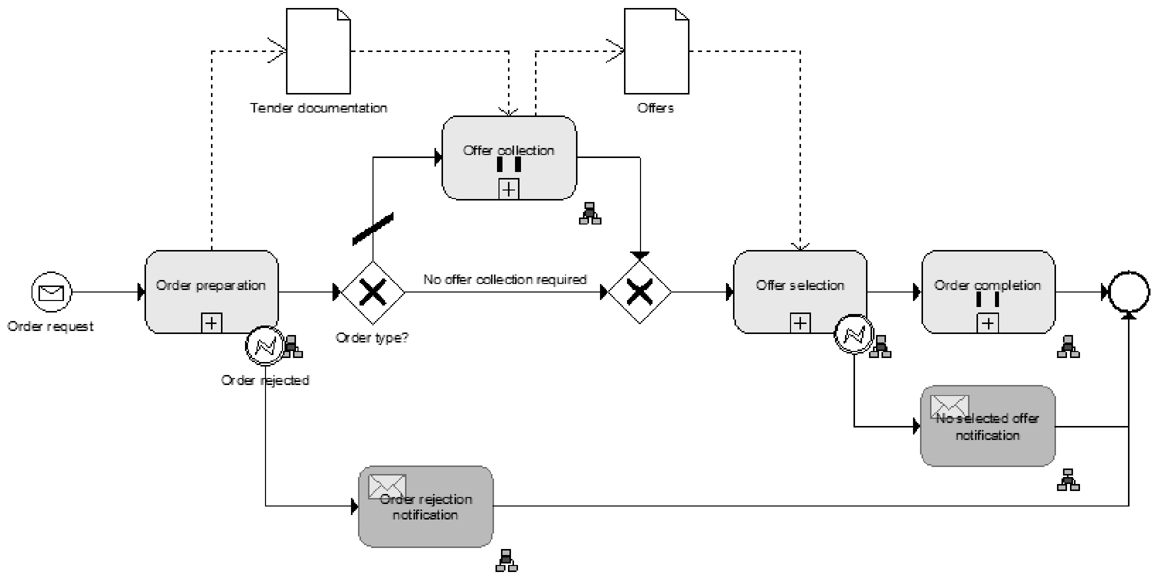

Fig. 5. Procurement process top-level BPMN diagram

On average, BPEL instances execute about 30 days, however, process instances for small orders can also be finalized in five days. On the other hand, BPEL process instances for very large orders can execute for a few months. For each new order, the corresponding process instance is started. Therefore, dozens of process instances are usually executed simultaneously. For each process instance, first the order is prepared (Order preparation sub-process) and then the tender documentation is sent to different suppliers, which can then send their offers within the specified time period (Offer collection sub-process). The list of suppliers is large and is managed through the company's ERP (Enterprise Resource Planning) system. For a specific process instance, there can be more than 40 suppliers that send their offer. The list of suppliers changes frequently as new suppliers are added, existing suppliers' data can change, certain suppliers may withdraw sent offers and some suppliers can also be temporarily or permanently suspended for various reasons (for example if they went out of business, or if they break state's strict laws and regulations).

Because process instances can take a very long time to complete, there are several steps in the process where it is necessary to synchronize suppliers' data. For example, when sending tender documentation, we always want to use the suppliers' latest contact information, as during the process execution (which can take a few months) this data can change. Similarly, before selecting the best offer, we want to retrieve the suppliers' latest credit rating and the latest list of references, as these are two of the most important decision parameters. On the other hand, some changes to suppliers' data (such as correction of typos in contact information) can also be done during the process instance execution. We want those changes to be automatically propagated back to the data source to avoid the need to make manual updates in two different systems. At certain process steps, we also want to check if there are any suppliers that have been suspended or have withdrawn their offers. In that case, the procurement process instance has to immediately eliminate those suppliers and their offers from the selection process. If we want to meet these requirements without using data-bound variables, we have to enforce data synchronization and checking in form of additional process steps.

Fig. 6 presents a simple process fragment, which we shall use to demonstrate how our proposed extensions can be used to reduce the BPEL code size and complexity.

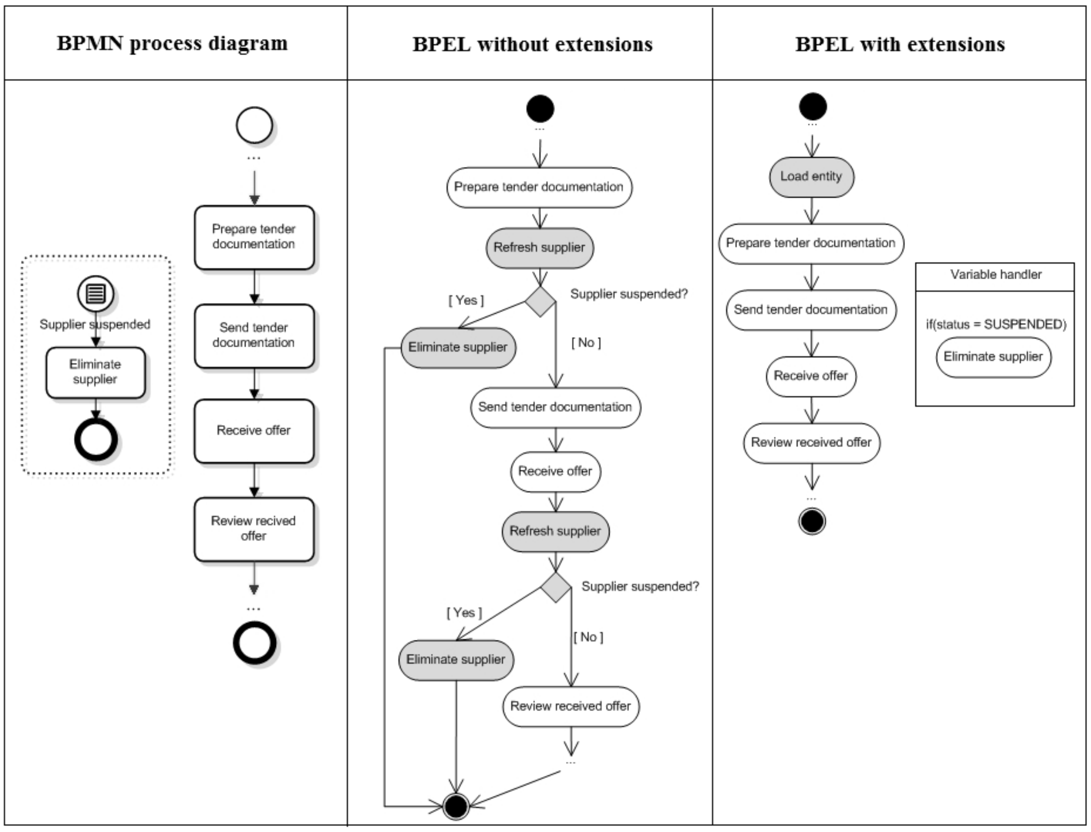

Fig. 6. A simplified fragment of the procurement process

The left lane on Fig. 6 contains a simplified fragment of a BPMN process diagram that we received as part of the specifications. It contains only four activities and an event sub-process. Following the requirements in the project documentation, all suspended suppliers have to be immediately eliminated from the selection process and the supplier's data always has to be refreshed before accessing it, as the data may change during the BPEL process execution. However, these data synchronization steps have not been modelled by the process analyst to improve the readability and to keep the model as simple as possible. The requirement regarding checking of supplier's status has been modelled using the event sub-process with conditional start event. The sub-process automatically triggers in case the supplier's status is changed to `SUSPENDED`, regardless of the active process step. The middle lane contains the UML activity diagram which demonstrates, how the same logic can be implemented using BPEL, without using our proposed extensions. We use the activity diagram only to present the main steps that have to be implemented in BPEL. The actual number of BPEL activities that are needed in order to implement the presented process steps can be higher. As BPEL does not provide a mechanism for automated data synchronization and for active monitoring of data in form of ECA rules, we have to add these steps (they are coloured with grey) as part of the process flow. If we would strictly follow the requirements, these additional steps would have to follow every step modelled in BPMN. However, this would result in a bad process design with a lot of code duplication. After a discussion with the process owner we identified the steps where the presented synchronization and status checking code is really necessary. In this simple example, there are two steps (before sending tender documentation and before reviewing the received offer) where supplier's data (such as contact information and status) has to be refreshed and where the checking of supplier's status has to be performed. However, adding those steps in the main process flow results in increased BPEL code size and complexity and reduces readability (please refer to the metric measurements results at the end of this section). The right lane presents, how the same process can be implemented using our proposed data-bound variables. We only have to add one activity (`<bpelx:loadEntity>`) to bind the dynamic variable to the data source. From that step on, the synchronization between the data-bound variable and the data source is performed automatically behind the scenes. The code that is used to handle the situation when supplier's status is changed is no longer duplicated in several steps in the process. Instead, it is placed inside the variable handler.

Without using our proposed extensions we may need up to eight or more BPEL activities to perform one such data synchronization and status checking. In our procurement process, there are 11 points where suppliers' data has to be necessarily synchronized and 5 points where we also have to perform supplier's status checking. The result of adding such data synchronization and data checking code at several steps in the process flow are increased size and complexity of the BPEL code.

The goal of our pilot project has been to test whether the same logic can be implemented using our extensions for data-bound variables. We had to develop an appropriate DAS to provide access to suppliers' data, where we were able to reuse most of the code from the existing service. We had to modify five of twelve BPEL processes that contain data synchronization code. In those processes, we first replaced existing service partner link with the new DAS partner link. Then we replaced all existing data synchronization code by adding data-bound variables and other related constructs.

7.3. Results

We wanted to experimentally evaluate positive effects of using our proposed solution, therefore we measured the size and the complexity of BPEL processes, using business process metrics [9]. To be able to measure the reduction in size and complexity, we used two process size metrics and two process complexity metrics:

- Number of activities (NOA) metric is used to measure the process size, by counting all activities in a process. We count only basic activities and omit structured activities, such as `<sequence>`, `<if>`, `<pick>`, `<while>`, `<flow>`, `<forEach>`, etc.
- Number of activities and control flow elements (NOAC) size metric counts the number of all basic activities and all control-flow elements (structured activities).
- Control flow complexity (CFC) metric calculates the number of splits and joins in a process. It is calculated using the following formula: $CFC_{BPEL}(P) = \sum C_{XOR} + \sum C_{OR} +$

$\sum C_{AND}$ [9]. C_{XOR} is complexity of XOR-split (equals to number of branches), C_{OR} is complexity of OR-split (equals to number of states that may arise from the execution of the split), and C_{AND} is complexity of AND-split (always equals 1). In BPEL, XOR-split is represented using `<if>` and `<pick>` activities, OR-split can be represented using BPEL links and AND-split is represented using the parallel `<forEach>` activity.

- McCabe's cyclomatic complexity metric (MCC) measures number of control paths through the process. MCC is defined to be $e - n + 2$, where e is the number of edges and n number of nodes in the control flow graph [9]. When measuring the MCC metric, we followed the MCC variant method for BPEL, as described in [24].

Measurement results for the presented procurement process and two other processes that we have also implemented are gathered in Table 1. The values present the reduction (in %) in size and complexity before and after using our proposed extensions. The results of the procurement process present an aggregation of all 12 BPEL processes that form a solution, although not all BPEL processes use the proposed extensions.

Table 1. Measurement results of three pilot projects

Metric	***Power distribution procurement process***	***Loan approval process***	***Airline ticket booking business process***	***Average [a] (by metric)***
NOA	-9,7 %	-9,2 %	-6,2 %	-8,4 %
NOAC	-8,9 %	-9,3 %	-5,8 %	-8,0 %
CFC	-6,1 %	-5,4 %	-3,7 %	-5,1 %
MCC	-3,9 %	-4,4 %	-4,1 %	-4,1 %
Average [a] (by project)	-7,2 %	-7,1 %	-5,0 %	-6,4 %

[a] Based on rounded values.

Using the presented metrics, we can see that all three processes are in any view smaller and less complex. The overall size and complexity of the projects using extensions have been reduced by 6,4%. Considering the process size metrics (NOA, NOAC), the size has been reduced by 8,2% on average. Similarly, the complexity metric (CFC, MCC) values show that the process complexity has been reduced by 4,6% on average. Furthermore, the resulting BPEL code is better-structured, as it contains less code duplication. Authors in [29,30] argue that process complexity directly impacts the readability. Therefore, we can assume that by reducing the complexity of the BPEL process the readability has also been improved. By introducing support for active monitoring of data with the use of ECA rules (concept that is already supported in BPMN) to BPEL, we have also reduced the semantic gap between BPMN and BPEL. With this, we have confirmed the real-world usability of the proposed solution.

8. Related work and discussion

The problem of using out-of-date data in BPEL processes has not been addressed yet in a way comparable to the approach proposed in this paper. However, several authors [4,25,26,28,36,44,45] have identified the problem of inefficient data access and manipulation in BPEL processes. Most of them claim that BPEL is not appropriate for data-intensive processes where large amount of data are passed throughout the process and between business partners. In order to improve performance and scalability, they propose several BPEL extensions which allow us to efficiently use data from remote data sources without the need to persist all data as part of the process instance state. In contrast, our approach focuses on providing a mechanism for automated refreshing and monitoring of BPEL variables, which not only ensures that we always use the latest version of important business data, but that we are also able to detect and react on all changes to data that were made by other applications.

Authors [25,26,36] propose several BPEL extensions and frameworks to enable efficient use of SQL at the business process orchestration level. Vrhovnik et al. [25,26] propose a framework for optimized data processing that allow tight integration of data processing capabilities into the process logic. In their proposed architecture, SQL inline support is included in the process choreography level and some data management activities are propagated directly to the underlying relational database. They mention so called SQL activities, which provide read and write access to BPEL variables that refer to remote database tables and support the assignment of query results to variables by reference, which improves performance. A similar approach that extends standard BPEL by introducing SQL snippet activities has already been implemented by IBM [43]. They introduce a new type of BPEL variables that can hold a reference to data in a remote database table and are called *set reference variables.* This approach improves performance and also indirectly addresses the problem of using outdated data in BPEL. However, the main drawback of such approach is that we have to code activities in the process logic which increases the complexity of process design and thus reduce its reusability. Another drawback is that we can only use data from relational data sources. In contrast, our proposed solution supports heterogeneous data source types and supports automated propagation of changes between local copy and remote data, without the need to code BPEL synchronization activities. We only have to declare data-bound variable and initialize it. Wieland et al. [4] also outline several drawbacks of passing large amounts of data by value in workflow-based applications. To address this challenge, they introduce the concept of pointers in BPEL. They introduce new type of BPEL variables, called reference variables. Reference variables present pointers to data in the data source. Their solution is not comparable to our proposed approach, as it does not provide a mechanism that would enable a BPEL process to automatically react on changes to remote data, as our variable handlers do.

There are also some existing extensions to BPEL that are somewhat similar to our approach. Apache ODE Group [12] has implemented BPEL extensions for external variables which allow us to map variables to remote data sources (JDBC and REST). Oracle [41] has also introduced new type of BPEL variables, called entity variables, that provide access remote relational data. However, those solutions do not support all types of data sources and also do not provide some important functionalities proposed in this paper. For example, they do not support the selection of the variable synchronization mode, the use of read-only variables and do not provide a mechanism to handle changes to remote version of data.

Some related work that also addresses the problem of data access and manipulation in BPEL, but from a perspective different than ours, can also be found on the use of Extract-Transform-Load (ETL) technology to define the data flow separate from the business control process flow. This approach has been proposed by Meier et al. [36]. Habich et al. [28,45] also identify that BPEL is not appropriate for data-intensive processes (like gene expression analysis processes), where a large amount of data is propagated between web service calls, due to performance and scalability issues. They claim that instead of transferring large amount of data through SOAP, technologies for data propagation like ETL or database replication should be applied. To overcome these problems, they introduce so called data-grey-box web services which allow us to use specialized data infrastructures for data propagation between web services and corresponding BPEL extensions, called BPEL data transitions. These proposed solutions, however, differ from our approach significantly, as their main goal is to improve performance and scalability and they do not address the problem of refreshing BPEL variables. However, some level of data refreshing can also be ensured during data propagation at the database level.

ECA rules for active monitoring of data have been used in many areas, including active databases [6,7], personalization and publish/subscribe technology [27,40], and specifying and implementing business processes [3,15,16,35,42]. Zaid et al. [15] propose the use of ECA rules to define conditions and corresponding recovery actions as part of their solution for supporting QoS-aware process execution. Baresi et al. [3] leverage ECA rules for defining recovery actions in their proposed solution for self-supervision BPEL processes. However, no author has yet proposed the use of ECA rules in business processes to actively monitor and handle changes to data in a way, comparable to our variable handlers.

The abovementioned approaches have numerous disadvantages when compared to our approach. Our proposed solution is the only solution that combines support for automated BPEL variable synchronization and support for active monitoring of changes to remote data. In addition, most of the presented approaches are limited to use data from relational data sources, only [4,12] also supports other types of data. Our solution allows us to use data from heterogeneous data sources (relational databases, XML databases, XML files, message queues, web services, etc), as long as we provide appropriate DAS. Another important advantage of our approach (in contrast to [25,26,36]) is that we do not need to code data synchronization steps, as this is done automatically by the BPEL engine behind the scenes. Finally, none of the existing solutions provide a mechanism that would support automated detection and handling of changes to remote data in BPEL code. On the other hand, by using our proposed variable handler, the BPEL process is able to proactively react on data changes, so that it becomes data-aware.

9. Conclusion

In this article, we have proposed a set of BPEL extensions that help the BPEL processes become data-aware. We have introduced a new type of BPEL variables, called data-bound variables. Data-bound variables are automatically synchronized with the data source and thus eliminate the need to implement data synchronization manually. To provide support for data-bound variables, we have proposed several extensions to BPEL and the use of appropriate DAS services.

To enable the use of data-bound variables, we have extended standard variable declarations. We have proposed two data synchronization modes: lazy and eager. We have introduced new activities for loading, creating and deleting entities. We have also introduced observed properties, observed property groups and a variable handler. Using this mechanism, the BPEL process is able to automatically adapt to changes to data that were made inside or outside the process scope.

To provide access to remote data, we have proposed the use of DAS services that expose CRUD data operations. We have also proposed DAS interface and introduced extensions to the BPEL partner link type to enable automated communication between the BPEL engine and DAS.

With the proposed approach, we have successfully solved the problem of using out-of-date data in BPEL processes. To demonstrate the usability of the extensions that we have proposed, we have implemented a prototype implementation on the open source ActiveBPEL engine and tested the solution in real-world environment, where we have confirmed the real-world usability of the proposed solution. We have also measured process size and complexity using various process metrics. The results show that when using our proposed solution, the overall size and complexity of the projects under test has been reduced by 6,4 % on average. We have also improved BPEL code readability and reduced semantic gap between BPMN process model and BPEL.

As part of our future work, we plan to introduce a new set of improvements. We plan to develop a set of XPath extension functions and extensions to `<assign>` activity that would allow us to load, create and delete remote entities simply using the `<assign>` activity. This would eliminate the need of using dedicated extension activities. We would also like to introduce a special fault management framework that would provide automated and declarative handling of DAS faults.

Appendix: BPEL extensions syntax specifications

Extensions to variable declaration:

```
<variables>
  <variable name="BPELVariableName"
            messageType="QName"?
            type="QName"?
            element="QName"?
            partnerLink="NCName"?
            bpelx:synchMode="lazy|eager"?
            bpelx:readOnly="yes|no"?>+

    <bpelx:observedProperties>?
      <bpelx:observedProperty name="NCName">+
        <query queryLanguage="anyURI"?>
           queryContent
        </query>
      </bpelx:observedProperty>
    </bpelx:observedProperties>
    <bpelx:observedPropertyGroups>?
      <bpelx:observedPropertyGroup name="NCName"
             observedProperties="NCName-list">+
      </bpelx:observedPropertyGroup>
    </bpelx:observedPropertyGroups>
    from-spec?
  </variable>
</variables>
```

`<bpelx:loadEntity>` for loading entities:

```
<extensionActivity>
  <bpelx:loadEntity variable="BPELVariableName"
                    standard-attributes>
    <bpelx:keyPart property="QName" queryLanguage="anyURI"?>+
       expression
    </bpelx:keyPart>
  </bpelx:loadEntity>
</extensionActivity>
```

`<bpelx:createEntity>` for creating entities:

```
<extensionActivity>
  <bpelx:createEntity variable="BPELVariableName"
                      standard-attributes>

  </bpelx:createEntity>
</extensionActivity>
```

`<bpelx:deleteEntity>` for deleting entities:

```
<extensionActivity>
  <bpelx:deleteEntity variable="BPELVariableName"
                      standard-attributes>

  </bpelx:deleteEntity>
</extensionActivity>
```

Variable handlers:

```
<bpelx:variableHandlers>
  <bpelx:onObservedPropertyChange variable="BPELVariableName"
         observedProperty="NCName" type="interrupting|non-interrupting"?>*
    <condition expressionLanguage="anyURI"?>?
      bool-expr
    </condition>
    activity
  </bpelx:onObservedPropertyChange>
  <bpelx:onObservedPropertyGroupChange variable="BPELVariableName"
         observedPropertyGroup="NCName" type="interrupting|non-interrupting"?>*
    activity
  </bpelx:onObservedPropertyGroupChange>
```

```
</bpelx:variableHandlers>
```

Extensions to `<partnerLinkType>`:

```
<wsdl:definitions name="NCName" targetNamespace="anyURI"...>
  ...
  <plnk:partnerLinkType name="NCName">
    <plnk:role name="NCName" portType="QName"/>
    <plnk:role name="NCName" portType="QName"/>?
    <plnkx:dataAccessServiceInstance name="NCName"
           dataGraph="QName"
           entity="QName"
           key="QName"
           keyParts="QName-list"
           createOpName="NCName"
           readOpName="NCName"
           updateOpName="NCName"
           deleteOpName="NCName"/>?
  </plnk:partnerLinkType>
  ...
</wsdl:definitions>
```

References

[1] Loshin D. Master Data Management. 1st ed. Burlington: Kaufmann Publishers; 2008.

[2] Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. In: ICSM'99 Proceedings of the IEEE international conference on Software Maintenance, 1999. p. 109-118.

[3] Baresi L, Guinea S. Self-Supervising BPEL Processes. IEEE Transactions on Software Engineering 2011; 37(2):247-263.

[4] Wieland M., Görlach K., Schumm D., Leymann F. Towards reference passing in web service and workflow-based applications. In: EDOC'09 Proceedings of the 13th IEEE international conference on Enterprise Object Computing, 2009. p. 89-98.

[5] Workflow patterns. URL: <http://www.workflowpatterns.com/patterns/data/routing/wdp39.php>.

[6] Widom J, Ceri S. Active Database Systems. San Mateo: Morgan Kaufmann Publishers; 1995.

[7] McCarthy D. The architecture of an active database management system. In: Proceedings of the ACM SIGMOD international conference on Management of data, 1989, p. 215-224.

[8] OMG. Business Process Modeling Notation (BPMN) Version 2.0, 2011, URL: <http://www.omg.org/spec/BPMN/>.

[9] Cardoso J, Mendling J, Neumann G, Reijers HA. A Discourse on Complexity of Process Models. In: Business Process Management Workshops, Springer Berlin Heidelberg, 2006, p. 117-128.

[10] Active Endpoints, ActiveBPEL engine, URL: <http://www.activebpel.org/>.

[11] Recker J, Mendling J. On the translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In: Proceedings of 18th International Conference on Advanced Information Systems Engineering, 2006, p. 521-532.

[12] Apache ODE Group, BPEL Extensions, 2010, URL: <http://ode.apache.org/bpel-extensions.html>.

[13] W3C. Web Services Description Language (WSDL) 1.1, W3C Note, 2001, URL: <http://www.w3.org/TR/wsdl>.

[14] W3C. Web Services Description Language (WSDL) Version 2.0, W3C Recommendation, 2007, URL: <http://www.w3.org/TR/wsdl20>.

[15] Zaid F, Berbner R, Steinmetz R. Leveraging the BPEL Event Model to Support QoS-aware Process Execution. In: Kommunikation in Verteilten Systemen 2009;9:93-104.

[16] Ying Z., Junliang C, Bo C, Yang Z. Using ECA rules to manage web service composition for multimedia conference system. In: Proceedings of the 2nd IEEE International Conference on Broadband Network & Multimedia Technology, 2009, p. 545-549.

[17] OASIS. Jordan D, Evdemon J. Web Services Business Process Execution Language Version 2.0. OASIS Standard, 2007, URL: <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>.

[18] Papamarkos G, Poulovassilis A, Wood PT. Event-Condition-Action Rule Languages for the Semantic Web. In: Lecture Notes in Computer Science, 2006, p. 855-864.

[19] Juric MB, Mathew B, Sarang P. Business Process Execution Language for Web Services. 2nd ed. Birmingham: Packt Publishing; 2006.

[20] OSOA. Service Data Objects For Java Specification Version 2.1.0, 2006, URL: <http://www.osoa.org/download/attachments/36/Java-SDO-Spec-v2.1.0-FINAL.pdf>.
[21] OSOA. Service Data Objects White Paper, 2007, URL: <http://www.osoa.org/download/attachments/287/SDO+V2.1+White+Paper.pdf?version=1>.
[22] Leff A, Rayfield JT. Programming models and Synchronization Techniques for Disconnected Business Applications. Advances in Computers 2006;67:85-130.
[23] Chen PP. The Entity-Relationship Model - Toward a Unified View of Data. ACM Transactions on Database Systems 1976;1:9-36.
[24] Havey M. Measuring SOA Complexity, 2010, URL: <http://www.packtpub.com/article/measuring-soa-complexity>.
[25] Vrhovnik M, Suhre O, Ewen S, Schwarz H. PGM/F: A Framework for the Optimization of Data Processing in Business Processes. In: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, 2008, p. 1584-1587.
[26] Vrhovnik M. An Approach to Optimize Data Processing in Business Processes. In: Proceedings of the 33rd International conference on very large databases, 2007, p. 615-626.
[27] Adi A, Botzer D, Etzion O, Hamam TY. Push technology personalization through event correlation. In: Proceedings of the 26th International conference on very large databases, 2000, p. 643-645.
[28] Habich D. et al. Data-aware SOA for Gene Expression Analysis Processes. In: 2007 IEEE Congress on Services, 2007, p. 138-145.
[29] Cardoso J. Process control-flow complexity metric: An empirical validation. In: 2006 IEEE International Conference on Services Computing, 2006, p. 167-173.
[30] Lassen KB, van der Aalst WM. Complexity metrics for Workflow nets. Information Software and Technology 2009;51:610-626.
[31] Resende L. Handling heterogeneous data sources in a SOA environment with service data objects (SDO). In: Proceedings of the 2007 ACM SIGMOD international conference on Management of data, 2007, p. 895-897.
[32] Alfarez JJ, Brogi A., Leite JA, Pereira LM. Computing Environment-Aware Agent Behaviours with Logic Program Updates. In: Selected papers from the 11th International Workshop on Logic Based Program Synthesis and Transformation, 2001, p. 216-232.
[33] Krizevnik M, Juric MB. Improved SOA persistence architectural model. ACM SIGSOFT Software Engineering Notes 2010;35(3).
[34] Pastrana JL, Pimentel E, Katrib M. QoS-enabled and self-adaptive connectors for Web Services composition and coordination. Computer Languages, Systems & Structures 2011;37(1);2-23.
[35] Liu A, Li Q, Huang L, Xiao M. A Declarative Approach to Enhancing the Reliability of BPEL Processes. In: Proceedings of the IEEE International Conference on Web Services, 2007, p. 272-279.
[36] Meier A, Mitschang B, Leymann F, Wolfson D. On Combining Busines Process Integration and ETL Technologies. In: Datenbankensysteme in Business, Technologie und Web, 11. Fachtagung des GI-Fachbereichs "Datenbanken und Informationsysteme", 2005, p. 533-546.
[37] Cui Z et al. A Functional Data Services Framework for Integrating Heterogeneous Data Sources. In: 32nd Annual IEEE Functional Computer Software and Applications Conference, 2008, p. 1150-1155.
[38] Brodsky S, Stockton M. SOA Programming model for implementing Web Services, Part 2: Simplified data access using Service Data Objects, 2005, URL: <http://www.ibm.com/developerworks/webservices/library /ws-soa-progmodel2.html>.
[39] Alfarez JJ, Banti F, Brogi A. An event-condition-action logic programming language. In: Proceedings of the 10th European Conference on Logics and Artificial Intelligence, 2006, p. 29-42.
[40] Bonifati A, Ceri S, Paraboschi S. Active rules for XML: A new paradigm for e-services, The International Journal on Very Large Databases, 2011:10:39-47.
[41] Oracle. Fusion Middleware Developer's Guide for Oracle SOA Suite 11g Release 1 (11.1.1), Part Number E10224-01, 2009, URL: <http://download.oracle.com/docs/cd/E12839_01/integration.1111/e10224/bp_manipdoc.htm>.
[42] Meng J, Su SY, Lam H, Helal A. Achieving Dynamic Inter-Organizational Workflow Management by Integrating Business Processes, Events and Rules. In: Proceedings of the 35th International Conference on System Sciences, 2002, pp. 102-112.
[43] IBM. WebSphere Integration Developer documentation, 2006, URL: <http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.wbit.help.nav.doc/topics/welcome.html>.
[44] Charfi A, Mezini M. An aspect-based process container for BPEL. In: Proceedings of the 1st workshop on Aspect oriented middleware development, 2005.
[45] Habich D et al. BPEL DT - Data-Aware Extension of BPEL to Support Data-Intensive Service Applications. Emerging Web Services Technology, 2008;2:111-128.

[46] Kopp O, Görlach K, Karastoyanova D, Leymann F, Reiter M, Schumm D et al. A Classification of BPEL Extensions. Journal of Systems Integration, 2011;4:3-28.

Vitae

Marcel Krizevnik is a researcher preparing his Ph.D. thesis. His main research areas are service oriented architecture and cloud computing. He has participated in several research and applicative projects that include business process optimization and consolidation, and the development of blueprints and pilot projects. He is also a member of SOA Competency Center and Cloud Computing Center. He is a co-author of WS-BPEL 2.0 for SOA Composite Applications with Oracle SOA Suite 11g book. He is an Oracle Service Oriented Architecture Infrastructure Implementation Certified Expert.

Matjaz B. Juric, Ph.D., is Full Professor at the University of Ljubljana and the head of SOA and Cloud Computing Competence Centre. He has authored 15 SOA and Java books, such as Business Process Driven SOA using BPMN and BPEL, 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, an Oracle ACE Director and a Java Champion.