Information and Software Technology xxx (2009) xxx–xxx

Contents lists available at ScienceDirect

Information and Software Technology

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

WS-BPEL Extensions for Versioning

Matjaz B. Juric [a,*], Ana Sasa [b], Ivan Rozman [a]

[a] University of Maribor, FERI, Institute of Informatics, Smetanova 17, SI-2000 Maribor, Slovenia
[b] University of Ljubljana, FRI, Information Systems Laboratory, Trzaska 25, SI-1000 Ljubljana, Slovenia

A R T I C L E I N F O

Article history:
Received 26 October 2008
Received in revised form 15 January 2009
Accepted 18 March 2009
Available online xxxx

Keywords:
Versioning
BPEL
SOA
Business processes

A B S T R A C T

This article proposes specific extensions for WS-BPEL (Business Process Execution Language) to support versioning of processes and partner links. It introduces new activities and extends existing activities, including partner links, *invoke*, *receive*, *import*, and *onmessage* activities. It proposes version-related extensions to variables and introduces version handlers. The proposed extensions represent a complete solution for process-level and scope-level versioning at development, deployment, and run-time. It also provides means to version applications that consist of several BPEL processes, and to put temporal constraints on versions. The proposed approach has been tested in real-world environment. It solves major challenges in BPEL versioning.

1. Introduction

In this article, we address the problem of versioning WS-BPEL (Business Process Execution Language) processes. BPEL has become the de-facto standard for orchestrating services in service oriented architecture (SOA). Versioning is an important topic in software development. It becomes even more important in SOA, where services are orchestrated into processes following the principle of loose coupling.

Services evolve over time in iterations, which result in new service versions. When orchestrated, BPEL processes need to be aware which versions of the services they use. Processes also evolve over time, which results in new process versions. Control over the versions becomes crucial, particularly in long-running processes, as we will explain later in this article.

Support for versioning in SOA is inadequate as neither BPEL nor other specifications such as WSDL (Web Services Description Language) and UDDI (Universal Description, Discovery, and Integration) provide explicit support for versions. In our previous work [22] we have proposed an approach to version services using WSDL and UDDI extensions. In this article, we focus on specific issues related to version support in BPEL. We propose extensions to BPEL to support versioning. We have addressed development, deployment, and run-time versioning of BPEL process themselves, and version control in orchestrations, where processes and services are consumed. We introduce new activities, extend existing activities, and introduce a version handler. We have designed the extensions for WS-BPEL version 2.0 using the standard language extension mechanism. We call the proposed extensions *WS-BPEL Extensions for Versioning.*

The article is organized as follows. In Section 2, we give a brief overview of BPEL. In Section 3, we explain the challenges in BPEL process versioning and highlight the current situation, which is inadequate. In Section 4, we outline our proposed solution, which consists of two parts, process versioning and partner link versioning. In Section 5, we describe the solution to version processes. We propose extension activities and attributes to the process and the scopes. We introduce a new handler type called version handler. In Section 6, we present version extensions to control the invocation of orchestrated (partner link) services and processes. We propose extensions for *invoke*, *receive*, *onmessage*, and *pick* activities and for the event handler. We also propose extensions for BPEL variables. In Section 7, we present the proof of concept, where we describe the implementation of the proposed extensions for versioning. In Section 8, we present related work and discuss the results. In Section 9, we give conclusions.

2. Brief overview of BPEL

BPEL is an OASIS standard [34] and has become the de-facto standard for service orchestration. BPEL is supported by the majority of SOA platforms and development tools [21]. It provides support for executable and abstract business processes. The current version is 2.0, which has been approved by OASIS in April 2007. BPEL 2.0 is an evolution of the previous version 1.1 and introduces several improvements, such as improved variable manipulation, enriched fault handling, improved correlation, local partner links,

* Corresponding author.
E-mail address: matjaz.juric@uni-mb.si (M.B. Juric).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.03.003

dynamic parallel flows, improved loop handling, and extension mechanism, which allows adding extensions to the BPEL language in a standardized way [20]. We use the BPEL extension mechanism to add versioning support. This way it is possible to implement versioning extensions for any available BPEL server.

Other extensions have been proposed for BPEL. BPEL Extensions for Sub-Processes [28] provide the means for the invocation of a business process as a sub-process of another business process, such that its lifecycle is coupled to the lifecycle of the parent process. BPEL Extension for People (BPEL4People) [1] addresses human interactions and introduces a new type of basic activity which uses human tasks as an implementation, and allows specifying tasks local to a process or use tasks defined outside of the process definition. This extension is based on the WS-HumanTask specification [2], which defines human tasks, including their properties, behavior, and a set of operations used to manipulate human tasks. AO4BPEL introduces aspect-oriented extensions to BPEL [7]. AdaptiveBPEL supports the development of differentiated and adaptive BPEL processes and is also based on the aspect-oriented concepts [12]. C-BPEL is an extension that incorporates context information and uses it for service composition [15]. BPEL4Chor is an extension for modeling choreographies [10].

3. Versioning in BPEL

Versioning is very important in software engineering [4], release planning [39] and particularly in SOA development [14,19]. In process orchestration, there are two aspects of versioning, which should be addressed: versioning of the process itself, and versioning of the partner links. Versioning of the process requires that we have means to manage and support the different process versions as a result of continuous development and process improvement. This includes the ability to deploy different versions of the same process and run them simultaneously. Here a special challenges are long-running processes, which can execute several days, weeks, or even months. When we deploy a new version, the existing instances have to be completed according to the old specification of the process. This means that several versions of the same process have to co-exist on the process server. Another challenge is clients who invoke processes. In SOA environments, we usually do not control all the clients a process has. In such cases, it is virtually impossible to upgrade all the clients at the same time as we deploy the new version of a process. This requires not only that different versions of the process co-exist on the same server, but also requires that the clients are able to invoke a specific version of the process.

Second aspect of versioning is the ability that a BPEL process invokes a specific version of a service or a process it consumes through a partner link. This includes the ability to discover version information and to bind to a specific version of a partner link. This means that we have to retain version information not only at development-time but also at run-time. Related to this requirement the BPEL process should also be able to handle situations where a specific version of a partner link service/process is unavailable and invoke another (possibly backward-compatible) version. Finally, the process should also be able to partner link services, which are not versioned.

BPEL currently does not provide any support for versioning. This is a serious drawback, which is addressed by some BPEL servers, such as IBM WebSphere Process Server [33] and Oracle BPEL Process Manager [35]. They provide very limited support for deployment-time versioning and allow deploying different processes under the same name, but with different version numbers. Usually two approaches are used. First is that the only accessible version is the latest version of the process, this is one that has been deployed most recently. Previous versions are only available to finish existing running process instances. The second approach is to publish different versions of the process under different endpoint URLs, which basically means that each process version is published as a separate endpoint. The problem of this approach is that there is no standard naming convention for version information. There is also no standard API to invoke different versions of the same process.

BPEL specification also does not address the problem of long-running processes, and what happens with them when a new version of a process is deployed. Existing instances of processes, which have been started using a previous version, should finish according to the old version specification. Related to this problem is the problem of upgrading services, which are consumed by a BPEL process. Suppose that we have started a long-running BPEL process. While the process is executing a service that this process invokes is upgraded to a new version. Should the already running process invoke the new or the old version of this service? BPEL does not provide any explicit support for invoking a specific version of a partner service. Currently the only solution for developers is to use different endpoint URL names for different service versions. This approach however is very inflexible, makes maintenance difficult, and at the same time considerably reduces the flexibility of such solutions. This approach is also limited to services that we control. For external services we cannot influence when they are upgraded. In such cases, the developers have to be familiar with the behavior of a specific BPEL server implementation to foresee the actual behavior of a running process where services have been upgraded in-between. This is a drawback particularly if we develop BPEL processes for servers from different vendors.

4. Proposed solution for BPEL versioning

The proposed *BPEL Extensions for Versioning* address all objectives, mentioned in the previous section. Our proposed approach provides support for development, deployment, and run-time versioning of processes. The versioning extensions also provide support for invoking specific versions of partner link services or processes. To achieve this we introduce specific extensions to BPEL. These include new activities that enable us to denote versions of processes and scopes (such as <*bpelx:version*>), and extensions to existing activities, including <*partnerLink*>, <*invoke*>, <*receive*>, and <*onmessage*> within <*pick*> and <*eventHandlers*>. We also propose a new handler, called version handler <*bpelx:versionHandlers*>, which is used to define the behavior of the process when a process client invokes a specific version of the process. Version handler can be used for the process or for the individual scopes.

Our proposed extensions enable BPEL developers to use process-level or scope-level versioning and introduce process bunches for versioning applications that consist of several BPEL processes. Versioning of processes and services has become an essential part of SOA development. Our solution makes processes version-aware. This way it simplifies planning for the fact that there will be many change requests [40]. Version-aware processes can consume version-aware services (as described in [22]) and version-unaware services. Our approach supports both version-aware and version-unaware process clients. Version-aware clients can select a process version and query for the version attributes. Our solution works best in environments where processes and involved partner link services use the proposed version extensions. This can be either within the same domain or between the domains. Our solution is also suitable for environments where processes consume partner link services for which we do not control their version cycle. In such cases, an appropriate approach is to develop version-aware façades in order to maximize benefits of the version extensions.

The proposed extensions use the standard BPEL extension mechanism <*extensionActivity*> and can therefore be implemented on any BPEL 2.0 compliant server. Listing 1 shows the declaration for the extensions, for which we use the *bpelx* namespace.

BPEL process versioning requires that we version the BPEL code and the WSDL interface of the process. In this article, we address the BPEL part. We have addressed versioning of WSDL interfaces in our previous work [22], where we have presented extensions to WSDL and UDDI to support versioning of all types of web service interfaces, irrespective of the implementation, including BPEL process interfaces. WSDL extensions for versioning enable to define multiple versions of a service interface. Versions can be denoted as *backward-compatible* or *backward-incompatible*. WSDL interface versioning can be done on the interface-level or on the operation-level. UDDI extensions for versioning extend the API for service providers and service consumers with version information. They introduce new keyed references for version-specific information. Version extended UDDI allows clients to query specific versions of services or processes. It provides traceability between version changes. For version-unaware clients it allows defining the current (default) version. It also provides mechanism for version deprecation.

In this article, we build upon the results of our previous work. However, we do not focus on versioning of interfaces, but on versioning of BPEL processes. Therefore, in the next sections we will first describe the proposed solution for versioning of BPEL processes. We will describe process-level and scope-level versioning and introduce version bunches. We will introduce the version handlers, which are used to handle scenarios where a specific version of a process is invoked. Version handlers can be defined at process or scope level. Finally, we will describe the extensions that allow invoking specific versions of partner link services or processes. We will describe extensions to partner links, *invoke*, *receive*, and *onmessage* activities. Let us first look at the versioning of BPEL processes.

5. BPEL process versioning

To version a BPEL process, we propose specific extensions that enable the development, deployment, and run-time versioning. We propose an extensional versioning solution with explicit versions. To uniquely identify each version of a BPEL process we introduce version identifiers (*vid*). The structure of the version identifier is defined by the versioning identifier scheme. A variety of version identifier schemes exist, such as numeric, date, year of release, and alphanumeric [47]. For example, a numeric version identifier schema might use the following structure: *major.minor*[.*build*[.*revision*]] or *major.minor*[.*maintenance*[.*build*]]. We support arbitrary version identifier schemas and define the version identifier (*vid*) as a custom simple XML schema type. To specify the required version identifier schema, we use a pattern restriction with a specific regular expression to match our required version identifier schema. This approach is compatible with the xADL 2.0 [17]. Listing 2 shows an example of a simple version identifier that consists of *major.minor* version numbers. We will use this version identifier in the rest of the article for simplicity purposes. In real world, we might want to use a more sophisticated identifier.

```
<xs:simpleType name="VersionID">
  <xs:restriction base="xs:string">
    <xs:pattern value="[0-9]{1,3}\.[0-9]{1,3}"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 2. Version identifier declaration example.

The rules how the various parts of a version identifier should be changed in relation to the type of changes in a process are defined by the versioning strategy. The definition of a versioning strategy is out of the scope of this article and is usually addressed as a part of the selected software development methodology or more specifically change management methodology.

Our approach to versioning supports different intents achieved through interface versioning. According to Conradi and Westfechtel [8] this can be revisions, which are intended to supersede its predecessor; variants, which are intended to co-exist; and version cooperation for multiple versions of processes. In the next subsections, we describe version declarations, process-level versioning, scope-level versioning, and version bunches.

5.1. Version declarations

To enable traceability of version changes and easier control over versions, we propose a version declaration section. Version declaration section can be nested within the BPEL implementation file, which we will describe first. It can also be placed into a separate file, which we will describe in Section 5.4.

Version declarations enable us to define the initial version (version 1.0 in our example in Listing 3). It also enables us to define the default version (version 2.0 in our example). Default version is useful for clients that bind to the process without specifying the required version. We can also declare a version as deprecated. This is useful for older versions for their gradual retirement and for migration of clients that use the older versions of the process. Particularly with public processes, where clients come outside of our organization boundaries, this is often the case. Clients can still use versions, which are marked as deprecated. Process providers however can retire deprecated processes without further notification. If a deprecated process is retired, clients will be unable to invoke the process version and will have to upgrade to a newer version. Clients should therefore interpret the deprecation notification as an explicit signal to upgrade to a newer version.

For each version, we declare the *type* of the version, which can be *backward-compatible* or *backward-incompatible*. This is useful for version-aware clients, which can select the most appropriate version programmatically. It is also useful for the tools, which can automate and simplify the development of versioned BPEL processes. We also declare the intent of the version, which can be *revision*, *variant*, or *cooperation*. This classification is based on [8], but

```
<process name="bookRatingProcess"
   targetNamespace="http://www.uni-mb.si/bookRating/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:bpelx="http://www.uni-mb.si/bpel/versioning/">

   <extensions>
      <extension namespace="http://www.uni-mb.si/bpel/versioning/"
                 mustUnderstand="yes" />
   </extensions>
...
```

Listing 1. Declaration of *BPEL Extensions for Versioning*.

```
<process name="bookRatingProcess"
   targetNamespace="http://www.uni-mb.si/bookRating/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:bpelx="http://www.uni-mb.si/bpel/versioning/">

  <bpelx:versions mode="processLevel">
    <bpelx:version vid="1.0">
      <bpelx:initialVersion/>
      <bpelx:deprecated/>
      <bpelx:validUntil>2008-06-30T10:00:00Z</bpelx:validUntil>
      <bpelx:versionInfo>Short description of version 1.0.</bpelx:versionInfo>
    </bpelx:version>
    <bpelx:version vid="1.1" type="backward-compatible" intent="variant">
      <bpelx:previousVersion vid="1.0"/>
      <bpelx:validFrom>2008-05-01T08:00:00Z</bpelx:validFrom>
      <bpelx:validUntil>2009-01-01T12:00:00Z</bpelx:validUntil>
      <bpelx:versionInfo>Short description of version 1.1.</bpelx:versionInfo>
    </bpelx:version>
    <bpelx:version vid="2.0" type="backward-incompatible" intent="revision">
      <bpelx:previousVersion vid="1.1"/>
      <bpelx:defaultVersion/>
      <bpelx:validFrom>2008-11-01T08:00:00Z</bpelx:validFrom>
      <bpelx:validUntil>2009-07-31T17:00:00Z</bpelx:validUntil>
      <bpelx:versionInfo>Short description of version 2.0.</bpelx:versionInfo>
    </bpelx:version>
  </bpelx:versions>

  <partnerLinks>
    …
  </partnerLinks>
  <variables>
    …
  </variables>
  <faultHandlers>
    …
  </faultHandlers>

  <!-- Process flow definition -->
</process>
```

Listing 3. Version declarations at the beginning of BPEL process.

can be adapted to cover other specific requirements. Intent information is used by the server at the deployment time to select the appropriate policy related to the process deployment and execution and the policy for handling the older versions of the process. In our example in Listing 3, version 1.1 is *backward-compatible* with 1.0. Version 2.0 in *backward incompatible* with version 2.0.

For each version we can include an optional <*wsdlx:versionInfo*> where we describe the changes in this specific version. Optionally, each version can also declare version validity in a certain date-time interval. To do this, it can specify *valid-from* and *valid-until* values, as shown on the example for version 2.0 (Listing 3). Temporal validity of a process can be used by the BPEL server to determine the default version of a process or to select the most appropriate version for version-unaware clients. More than one version of a process can be valid at a certain time. If *valid-from* and *valid-until* are not specified the BPEL run-time environment does not place any temporal limitations on the version. For deprecated processes, temporal validity specifies until when the process will be available and the clients have to upgrade by the *valid-until* date. Process clients can programmatically query version information, such as version identifier, version info, type, intent, and other to decide which process version to use.

Tools can be used to simplify and automate version declarations, track backward compatibility, and integrate version declarations with servers and deployment descriptors. Such tools, which are supported by our approach, can considerably simplify the version related development.

In Listing 3, we can also see that we can define the version mode. We can use *process-level*, *scope-level*, or *process-bunch* versioning. We will describe them in the following subsections.

5.2. Process-level versioning

Process-level versioning is the coarse-grained approach. It allows us to define a separate process flow definition for each version. Process-level versioning is useful for larger version changes, particularly if they are backwards incompatible. To use process-level versioning we place a <*bpelx:version*> activity as the main activity of the process and define the version identifier (*vid*). Within the <*bpelx:version*> we declare the usual BPEL elements, including imports, partner links, message exchanges, variables, correlation sets, fault handlers, event handlers, and the process flow itself (usually starting with a <*sequence*> or a <*flow*> activity).

All BPEL elements (imports, partner links, message exchanges, variables, etc.) defined within the <*bpelx:version*> are specific to this version. We can also define imports, partner links, message exchanges, variables, correlation sets, fault handlers, and event handlers outside the <*bpelx:version*> declaration. In this case, the defined elements are version-neutral and applicable to all versions. A specific version can declare its own elements (for example fault handlers), which override the version-neutral declarations. Such approach is useful in real-world scenarios, where we have observed that different versions of a BPEL process often share imports, fault and event handlers, partner links, and some variable declarations. Example is shown in Listing 4.

5.3. New import type

If we deal with large processes, the above-described approach might result in very large source *.bpel* files, which are difficult to maintain. To solve this problem, we introduce a new import type, which allows us to break a single BPEL process into several files, organized per versions, and import them.

Originally, BPEL provides two types of imports: imports for XML Schemas (http://www.w3.org/2001/XMLSchema) and imports for WSDLs (http://schemas.xmlsoap.org/wsdl/). Our extension allows importing BPEL processes. To import processes the *importType* attribute must be set to http://docs.oasis-open.org/wsbpel/2.0/process/executable or http://docs.oasis-open.org/wsbpel/2.0/process/abstract, for executable and abstract BPEL processes, respectively. Example is shown in Listing 5, where the *Book Rating* process version 1.0 has been distributed into two files. The semantics of BPEL import is identical to the semantics of XML Schema and WSDL imports.

5.4. Separating version information and process implementation

Separating version information and process implementation is useful in many scenarios and can contribute to better separation of concerns. For example, in larger development teams the actual BPEL developer might not be aware of the overall version complexity. Separation of version information and implementation also allows developing BPEL code without extensions and maintaining version extensions as separate artifacts.

Our approach supports such separation. It specifies version information using the same syntax as used for abstract business processes. This way, the process version declaration file maintains the BPEL syntax and can be used in BPEL development environments without modifications. To link the version information with the BPEL process implementation, we use the extended *<import>* activity. An example is shown in Listing 6, where the version information is defined as a part of the abstract process, which uses *<import>* to link the version information with the actual BPEL implementation code for each version.

5.5. Versioning of processes bunches

To version applications, which consist of several BPEL processes we introduce process bunches that allow to specify the version lifecycle of several interrelated processes (process bunches). To maintain the syntax we propose to specify process bunches as abstract processes and use the extended *<import>* activity to define the executable processes, of which an application (bunch) consists of. The example is shown in Listing 7 and shows a Book application, which

```
<process name="bookRatingProcess"
   targetNamespace="http://www.uni-mb.si/bookRating/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:bpelx="http://www.uni-mb.si/bpel/versioning/">

  <bpelx:versions mode="processLevel">
    <!-- See Listing 3 -->
  </bpelx:versions>

  <!-- These elements are valid for all versions -->
  <import .../>
  <partnerLinks>...</partnerLinks>
  <messageExchanges>...</messageExchanges>
  <variables>...</variables>
  <correlationSets>...</correlationSets>
  <faultHandlers>...</faultHandlers>
  <eventHandlers>...</eventHandlers>

  <bpelx:version vid="1.0">
    <!-- These elements are for this version only -->
    <import .../>
    <partnerLinks>...</partnerLinks>
    <messageExchanges>...</messageExchanges>
    <variables>...</variables>
    <correlationSets>...</correlationSets>
    <faultHandlers>...</faultHandlers>
    <eventHandlers>...</eventHandlers>
    <sequence>
      <!-- Process flow is defined here -->
      ...
    </sequence>
  </bpelx:version>

  <bpelx:version vid="1.1">
    ...
  </bpelx:version>

  <bpelx:version vid="2.0">
    ...
  </bpelx:version>
</process>
```

Listing 4. Process-level versioning.

```
<process name="bookRatingProcess"
   targetNamespace="http://www.uni-mb.si/bookRating/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:bpelx="http://www.uni-mb.si/bpel/versioning/">

  <bpelx:versions mode="processLevel">
    <!-- See Listing 3 -->
  </bpelx:versions>

  <bpelx:version vid="1.0">
    <!-- Some BPEL code -->
    ...
    <!—Import first part of the process -->
    <import importType="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
      location="bookRatingProcess_part1.bpel"
      namespace="http://www.uni-mb.si/bookRating/" />
    ...
    <!-- Some BPEL code -->
    ...
    <!-- Import second part of the process -->
    <import importType="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
      location="bookRatingProcess_part2.bpel"
      namespace="http://www.uni-mb.si/bookRating/" />
    ...
    <!-- Some BPEL code -->
  </bpelx:version>

  <bpelx:version vid="1.1">
    ...
  </bpelx:version>

  <bpelx:version vid="2.0">
    ...
  </bpelx:version>
</process>
```

Listing 5. New import type.

consists of three BPEL processes, *Book Rating*, *Book Purchase*, and *Book Delivery*. The process bunch declares three versions, initial version 1.0 that is now deprecated, version 1.1, and default version 2.0.

5.6. Scope-level versioning

For fine-grained version control, we propose scope-level versioning. A BPEL process consists of several scopes, which can be nested to an arbitrary level. Scope-level versioning allows to version each process scope individually. We can define which scope versions constitute a specific process version. This approach is flexible and particularly appropriate for complex processes as it allows to version process scopes individually. It also allows reusing different scope versions among different processes versions. Scope-level versioning however requires more detailed version planning and increases the complexity, as each scope is versioned individually.

To version scopes, we use the <*bpelx:version*> extension activity, which should be used as the primary scope activity. A scope can declare the associated <*partnerLinks*>, <*messageExchanges*>, <*variables*>, <*correlationSets*>, <*faultHandlers*>, <*compensationHandler*>, <*terminationHandler*>, and <*eventHandlers*> outside or inside the <*bpelx:version*> activity. If declared outside the <*bpelx:version*> extension activity, the declarations are version-neutral and are applied to all scope versions. This is particularly useful for partner links, fault and compensation handlers, and for some variables. If declared inside, they are specific to the version and override the version-neutral declarations. This is shown in Listing 8.

Scopes can be declared as *backward-compatible* or *backward-incompatible* using the *type* attribute. Backward compatibility is useful for developers planning a specific process version. This information is used by the server for selecting the appropriate deployment policy. Backward compatibility can also be used by the tools, which can automate and simplify version management of scopes. If backward compatibility is not specified the default is *backward-incompatible*.

To explicitly reuse a scope from a previous version, we introduce a new <*bpelx:useVersion*> activity. Example is shown in Listing 8, where in *scope1* in version 1.1 we reuse version 1.0. This simplifies version management and makes it more transparent. It improves maintainability and reduces the need to copy and paste the same code parts. <*bpelx:useVersion*> activity automatically implies *backward-compatible* versions.

Scopes that do not change between different versions of a process do not need to be versioned. Non-versioned scopes are equal to all process versions. A specific BPEL process version is composed of versioned scopes with the specific version identifier and of non-versioned scopes. Versioned scope definitions always override non-versioned scope definitions.

We can override the default behavior by explicitly specifying the scope versions that should be included in a specific version of a BPEL process. We specify them within the <*bpelx:versions*> element. We introduce a <*bpelx:scope*> activity that we use to specify scope versions that constitute a specific version of BPEL process. Example is shown in Listing 9.

It is the responsibility of the developer to provide a specification of the scope versions that constitute a complete process, which is syntactically and semantically correct. In practice, this approach can be supplemented using tools that will automatically compare scopes and identify the parts which are common and which differ among versions. Tools can also help in merging versions and declaring version information. Such tools can considerably

```
<process name="bookRatingProcess"
   targetNamespace="http://www.uni-mb.si/bookRating/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
   xmlns:bpelx="http://www.uni-mb.si/bpel/versioning/">

  <bpelx:versions mode="processLevel">
    <bpelx:version vid="1.0">
      <bpelx:initialVersion/>
      <bpelx:deprecated/>
      <bpelx:validUntil>2008-06-30T10:00:00Z</bpelx:validUntil>
      <bpelx:versionInfo>Short description of version 1.0.</bpelx:versionInfo>
      <import importType="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        location="bookRatingProcess.1.0.bpel"
        namespace="http://www.uni-mb.si/bookRating/" />
    </bpelx:version>
    <bpelx:version vid="1.1" type="backward-compatible" intent="variant">
      <bpelx:previousVersion vid="1.0"/>
      <bpelx:validFrom>2008-05-01T08:00:00Z</bpelx:validFrom>
      <bpelx:validUntil>2009-01-01T12:00:00Z</bpelx:validUntil>
      <bpelx:versionInfo>Short description of version 1.1.</bpelx:versionInfo>
      <import importType="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        location="bookRatingProcess.1.1.bpel"
        namespace="http://www.uni-mb.si/bookRating/" />
    </bpelx:version>
    <bpelx:version vid="2.0" type="backward-incompatible" intent="revision">
      <bpelx:previousVersion vid="1.1"/>
      <bpelx:defaultVersion/>
      <bpelx:validFrom>2008-11-01T08:00:00Z</bpelx:validFrom>
      <bpelx:validUntil>2009-07-31T17:00:00Z</bpelx:validUntil>
      <bpelx:versionInfo>Short description of version 2.0.</bpelx:versionInfo>
      <import importType="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        location="bookRatingProcess.2.0.bpel"
        namespace="http://www.uni-mb.si/bookRating/" />
    </bpelx:version>
  </bpelx:versions>
</process>
```

Listing 6. Separating version information and BPEL implementation.

alleviate the burden of versioning and simplify the work for the developers. Our approach supports usage of such tools.

5.7. Version handler

To enable better control over versions and their management, we propose a version handler. The purpose of the version handler is to manage incoming calls from process clients, which invoke a specific version of the process. Let us suppose that we have a BPEL process that has several versions, which have been implemented over time. Versions 1.0, 1.1, and 1.2 are semantically compatible, but require schema transformations. To simplify maintenance, we have decided to abandon versions 1.0 and 1.1. Normally this would require updating all clients that still use versions 1.0 and 1.1. With version handler however, we can route all requests that come for versions 1.0 and 1.1 to version 1.2 and do the necessary pre-processing and post-processing transformations. Listing 10 shows an example, where we do a pre-processing XSLT transformation to adapt the version 1.0 schema to version 1.2, then call version 1.2, and finally invoke a XSLT transformation to adapt the output of version 1.2 to the schema used by version 1.0.

Version handler is defined using a new <*bpelx:versionHandlers*> activity. We nest specific <*bpelx:onVersion*> activities within the version handler. Version handler can be applied to a process or a scope, similarly as fault or event handler. Version handler defines the behavior that is executed when a specific version of a process or scope is invoked. This enables us to gain better control about versions and their lifecycle. Typical use cases for version handler include migration of versions, version deprecation, and version retirement. Very often, it will make sense to invoke another version of a process or a scope within the version handler. To do this we introduce a new activity, <*bpelx:invokeVersion*>, which can be used within version handler only. For this activity, we specify the *name* of the process or scope, the version identifier of the desired version that we invoke, and the optional *variable* name that is used to store the result of the invoked process or scope. Variable with the result is usually used to transform the result to match the requirements of a specific version.

Sometimes it makes sense to log requests that came for various versions. For example, let us suppose that we have deprecated version 1.0 some time ago and would like to retire it now. Using a version handler, we can log all requests that have come for version 1.0 and use the log information to notify the clients that they should upgrade to a newer version, because version 1.0 is retired. To log requests we introduce a special activity, <*bpelx:logRequest/*>, which can be used within version handler only. The BPEL engine has to provide means to configure where the log will actually be written and what format it will use.

The clients, who try to invoke the retired process version will get a fault notification with the explanation that this version of the process is retired and that they should upgrade to a newer version. They will also get the information, which version is the default version, if we have marked one version with the <*bpelx:defaultVersion*> or with temporal validity. In Listing 11 we show an example, where we have retired the version 1.0 and log the incoming requests to keep track of clients that still use the retired version.

With this, we have concluded our discussion on BPEL process versioning. In the next section, we will discuss how to invoke specific versions of consumed processes and services (partner links).

```
<process name="bookApp"
   targetNamespace="http://www.uni-mb.si/bookRating/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
   xmlns:bpelx="http://www.uni-mb.si/bpel/versioning/">

  <bpelx:versions mode="processBunch">
    <bpelx:version vid="1.0">
      <bpelx:initialVersion/>
      <bpelx:deprecated/>
      <bpelx:versionInfo>Short description of version 1.0.</bpelx:versionInfo>
      <import importType="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        location="bookRatingProcess.1.0.bpel"
        namespace="http://www.uni-mb.si/bookRating/" />
      <import importType="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        location="bookPurchaseProcess.1.0.bpel"
        namespace="http://www.uni-mb.si/bookPurchase/" />
      <import importType="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
        location="bookDeliveryProcess.1.0.bpel"
        namespace="http://www.uni-mb.si/bookDelivery/" />
    </bpelx:version>

    <bpelx:version vid="1.1" type="backward-compatible" intent="variant">
      <bpelx:previousVersion vid="1.0"/>
      <bpelx:versionInfo>Short description of version 1.1.</bpelx:versionInfo>
      <import .../>
      <import .../>
      <import .../>
    </bpelx:version>

    <bpelx:version vid="2.0" type="backward-incompatible" intent="revision">
      <bpelx:previousVersion vid="1.1"/>
      <bpelx:defaultVersion/>
      <bpelx:versionInfo>Short description of version 2.0.</bpelx:versionInfo>
      <import .../>
      <import .../>
      <import .../>
    </bpelx:version>
  </bpelx:versions>
</process>
```

Listing 7. Versioning of process bunches.

6. Invoking specific partner link versions

A BPEL process is an orchestration of services and/or processes. The orchestrated services and processes are specified as partner links. For example, a *Book Purchase* process might invoke a Book Price service, a Credit Card service, and a *Book Delivery* process. Book Price, Credit Card, and *Book Delivery* are partner links.

Very often partner links need to be versioned. For example, several versions of the Credit Card service might exist and several versions of the *Book Delivery* process might exist. The *Book Purchase* process might want to bind to specific versions of partner link services and processes, or it might want to use the default versions.

Currently BPEL does not provide support to bind to specific versions of partner links. Therefore, in this section we will describe the proposed extensions that allow a BPEL process to invoke partner links by specifying version information. First, we will describe extensions to <*partnerLink*> activity. Next, we will describe extensions to <*invoke*>, <*receive*>, <*pick*>, and <*onmessage*> activities. Finally, we will describe the extensions to variables.

6.1. Partner link extension

For versioned partner links, we propose to specify the version identifier at the same time as declaring the partner link. This is done within the <*partnerLink*> activity on the process or scope level. We introduce an extension attribute, *bpelx:vid* to specify the required version of the partner link. Listing 12 shows example for Book Price partner link.

If we would like to use the default version of a partner link, we specify the *bpelx:defaultVersion* attribute, as shown for the Credit Card partner link. This way our process will always use the default version of the partner service.

Sometimes the BPEL process and the partner links have the same version lifecycle, for example if they are part of the same application. In this case, the BPEL process version should be the same as the version of partner links. This is achieved by *bpelx:implicitVersion* attribute, as shown in Listing 12 for the *Book Delivery* partner link. If the partner link is not versioned, we simply do not specify the version extension attribute.

6.2. Extensions for invocation activities

Once we have defined the version of the partner link, all interactions with the partner link through <*invoke*>, <*receive*>, <*pick*>, and other activities are related to the version that is specified in the partner link. In the majority of cases, this is adequate. However, in some special occasions we might want to directly influence the versions, which are invoked from the above-mentioned activities. For example, let us suppose that the default version of a service that is invoked is either not available, or does not work. In this case, a fault handler is activated to handle the exception. Instead of failing, our process will invoke another version of the same service, which is available and does work.

```
<process name="bookRatingProcess"
   targetNamespace="http://www.uni-mb.si/bookRating/"
   xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
   xmlns:bpelx="http://www.uni-mb.si/bpel/versioning/">

  <!-- These elements are for all versions -->
  <import .../>
  <partnerLinks>...</partnerLinks>
  <messageExchanges>...</messageExchanges>
  <variables>...</variables>
  <correlationSets>...</correlationSets>
  <faultHandlers>...</faultHandlers>
  <eventHandlers>...</eventHandlers>

  <sequence>
    <receive .../>

    <scope name="scope1">
      <!-- These elements are for all versions -->
      <variables>...</variables>
      <partnerLinks>...</partnerLinks>
      <messageExchanges>...</messageExchanges>
      <correlationSets>...</correlationSets>
      <eventHandlers>...</eventHandlers>
      <faultHandlers>...</faultHandlers>
      <compensationHandler>...</compensationHandler>
      <terminationHandler>...</terminationHandler>

      <bpelx:version vid="1.0">
        <!-- These elements are for this version only -->
        <variables>...</variables>
        <partnerLinks>...</partnerLinks>
        <messageExchanges>...</messageExchanges>
        <correlationSets>...</correlationSets>
        <eventHandlers>...</eventHandlers>
        <faultHandlers>...</faultHandlers>
        <compensationHandler>...</compensationHandler>
        <terminationHandler>...</terminationHandler>
        <sequence>
          <!-- Scope flow is defined here -->
          ...
        </sequence>
      </bpelx:version>

      <bpelx:version vid="1.1" type="backward-compatible">
        <bpelx:useVersion vid="1.0"/>
      </bpelx:version>

      <bpelx:version vid="2.0" type="backward-incompatible">
        ...
      </bpelx:version>
    </scope>

    <scope name="scope2">
      <!-- Versioned in a similar way as Scope1 -->
      ...
    </scope>

    <scope name="scope3">
      <!-- This scope is not versioned -->
      ...
    </scope>
  </sequence>
</process>
```

Listing 8. Scope-level versioning.

To achieve this, our proposal allows to use *bpelx:vid*, *bpelx:defaultVersion*, and *bpelx:implicitVersion* attributes directly on <*invoke*>, <*receive*>, and <*onmessage*> activities within <*pick*> and <*eventHandlers*>. Listing 13 shows an example, where we invoke the Book Service version 2.3 instead of version 2.5, as defined by partner link.

Invoking a partner link version other than specified in the <*partnerLink*> declaration places some responsibilities on the BPEL

```
<bpelx:versions mode="scopeLevel">
  ...
  <bpelx:version vid="1.5">
    <bpelx:previousVersion vid="1.1"/>
    <bpelx:defaultVersion/>
    <bpelx:scope name="scope1" vid="1.1"/>
    <bpelx:scope name="scope2" vid="1.0"/>
    <bpelx:scope name="scope3"/>  <!-- Scope3 is not versioned -->
  </bpelx:version>
  ...
</bpelx:versions>
```

Listing 9. Explicit specification of versions for scope-level versioning.

```
<bpelx:versionHandlers>
   <bpelx:onVersion vid="1.0">
      <!-- Pre-processing -->
      <assign>
        <copy>
          <from>
            bpel:doXslTransform("urn:stylesheets:InpVer10to12.xsl", $Input10)
          </from>
          <to variable="Input12" />
        </copy>
      </assign>
      <bpelx:invokeVersion name="MyProcess" vid="1.2" variable="Output12"/>
      <!-- Post-processing -->
      <assign>
        <copy>
          <from>
            bpel:doXslTransform("urn:stylesheets:OutVer12to10.xsl", $Output12)
          </from>
          <to variable="Output10" />
        </copy>
      </assign>
   </bpelx:onVersion>

   <bpelx:onVersion vid="1.1">
      <!-- Pre-processing -->
      <bpelx:invokeVersion name="MyProcess" vid="1.2" variable="Output12"/>
      <!-- Post-processing -->
   </bpelx:onVersion>
</bpelx:versionHandlers>
```

Listing 10. Version handler for routing requests to a different version of a process.

```
<bpelx:versionHandlers>
   <bpelx:onVersion vid="1.0">
      <bpelx:logRequest/>
      <bpelx:retired/>
   </bpelx:onVersion>
</bpelx:versionHandlers>
```

Listing 11. Log requests and retire versions with version handler.

server and on the developer of the BPEL process. The BPEL server has to dynamically bind to a different service version. It has to resolve the version-aware WSDL and bind to the corresponding URL on which a different partner link version is published. To achieve this it has to use the required version mapping to endpoint URLs, as described in [22]. As all versions of a service are described within a single version-aware WSDL, no changes to the <*import*> activity are required.

The process developer has to be aware that a different version might or might not be compatible with the version specified in the partner link. Therefore, the developer has to declare the corresponding input and output variables that can be used for the invocation and has to foresee the semantic differences in operations

```
<partnerLinks>
   <partnerLink name="BookPrice"
      partnerLinkType="tns:BookPriceLT"
      myRole="BookPriceRequester"
      partnerRole="BookPriceService"
      bpelx:vid="2.5" />

   <partnerLink name="CreditCard"
      partnerLinkType="tns:CreditCardLT"
      myRole="CreditCardRequester"
      partnerRole="CreditCardService"
      bpelx:defaultVersion="yes" />

   <partnerLink name="BookDelivery"
      partnerLinkType="tns:BookDeliveryLT"
      myRole="BookDeliveryRequester"
      partnerRole="BookDeliveryService"
      bpelx:implicitVersion="yes" />
</partnerLinks>
```

Listing 12. Partner link extensions for versioning.

between versions (if they exist). To support this, specific extensions to BPEL variables are required.

```
<invoke partnerLink="BookPrice"
   portType="tns:BookPricePT"
   operation="getBookPrice"
   inputVariable="BookPriceRequest"
   bpelx:vid="2.3" />
```

Listing 13. Explicit version specification in <*invoke*> activity.

6.3. Extensions for variables

Variables in BPEL can be of three kinds: WSDL message types, XML Schema types, or XML Schema elements. By default, the versions of message types and in-line schemas are determined by the version, specified in the partner link. The version is resolved from the partner link type porttype declaration, from which the messages can be obtained.

If however explicit version is specified in the invocation activities, as described in previous section, a developer might need to declare variables using message types or elements from different interface version. Therefore we have extended the <*variable*> declaration with the *bpelx:vid* attribute. For example, to be able to invoke the *getBookPrice* operation using version 2.3 of the Book Price partner link, we need to declare a *BookPriceRequest* variable that uses the WSDL message type from the 2.3 interface. The example is shown in Listing 14.

With this, we have concluded our discussion on invoking specific versions of partner links.

7. Proof of concept

The *WS-BPEL Extensions for Versioning* has been verified on a real-world SOA project in a large power distribution company. In this company, several BPEL processes were implemented. Some of them have been long-running, especially the procurement process. As the power distribution company is state owned, the procurement process is very complex. It has several automated activities, such as creating order form, comparing the offers, creating a draft contract, performing updates to the accounting system, etc. It also has several human activities, such as negotiating contract price and conditions, approving order form, selecting best offer, approving final selection, etc. For each new order the corresponding BPEL process is started. This means that several instances of the procurement BPEL process are executed simultaneously. Usually, this is done once per month, however for urgent orders it is also done on request. On average approximately three quarters of the BPEL instances execute from 27 to 31 days. Some BPEL process instances for the smaller orders can be finalized as soon as in five days. On the other hand, BPEL process instances for the larger orders can execute for a few months.

Automated activities in BPEL process have been implemented as partner link services and, some of them, as BPEL sub-processes. Most of the partner links in the procurement process are internal. However, the process also uses some external partner links (such as publishing the order request on a central statewide portal, which is required by the law). In total, the procurement BPEL process has 27 partner links to the internal services, 3 partner links to the external services, and 18 different human tasks. 25 partner link services are not exclusive to this process, but are reused by the other processes as well.

```
<variables>
   <variable name="BookPriceRequest"
      messageType="tns:BookPriceRequestMessage"
      bpelx:vid="2.3" />
</variables>
```

Listing 14. Extensions for variables.

Due to dynamic nature of the company and the requirements, there have been frequent changes to the partner link services and the process itself, usually several times per month. Using existing BPEL servers, we have faced several problems related to versioning. We have run into the problem of deploying new process versions while some instances (using the older version declarations) have still been running. Managing different versions of the processes has been crucial, as the processes have been mission critical. None of the available BPEL servers has provided a sound solution to this problem.

As there were so many changes to the processes, we were faced with very many versions of the same process. After first 6 months, the number of versions has grown considerably and we started to observe that a lot of time was spent on version management and maintenance. In addition, considerable server resources have been used, as each process version has been deployed as a separate process. None of the tested BPEL servers provided a good way to migrate, deprecate, and retire versions.

BPEL processes consumed several services, which have changed too. When we deployed a new version of the service, all processes started to use this new version immediately. This was true also for the process instances that had been started before the service has been upgraded. The requirement for such process instances has been to use the old versions of the service. The only solution for this problem (without version extensions) was to use different endpoint URLs for each service and process version. As the number of consumed services was high, this required manual updating of partner link references, which was very time consuming and error-prone.

To solve these and some other problems, mentioned in this article, we have designed the versioning extensions. We have implemented the proposed *WS-BPEL Extensions for Versioning* on the open source BPEL server Apache ODE 1.1 [3]. There are two possible approaches for *WS-BPEL Extensions for Versioning* implementation: native implementation of the extensions on the BPEL engine, or implementation of the extensions as a pre-processor to generate plain BPEL out of versioned BPEL code. Both approaches require modifications to the BPEL engine in respect to invoking versioned processes. We have chosen the native approach as it results in a cleaner integration of the version information with the process deployment and run-time environment and consumes less server resources. The prototype implementation has taken seven person-months and has been done by a well-skilled development group.

The execution of versioned processes does not place any mentionable additional run-time overhead on the BPEL server. The additional run-time overhead of the version extensions is limited to the selection of the appropriate process version when invoking the process, checking the version constraints, and binding with the version-aware services. All these tasks are part of the process initialization and do not affect the actual execution of the process instances. All other version-specific details are resolved at the deployment time. *WS-BPEL Extensions for Versioning* does not influence the scalability of processes, which is determined by the architecture of the specific BPEL engine.

After initial three month testing of the versioning extensions in the real-world environment of the power distribution company, we can report that we have successfully solved all the above-mentioned problems. Explicit version information of processes and services has solved the problem of version upgrades and their management. Version extensions for partner links together with BPEL process versioning have solved the problem of different BPEL process versions. In particular, our solution has solved the problem of upgrading process versions while existing process instances have been running. Our solution has also solved the problem of upgrading partner link services while process instances have been running. Our solution assures that always the right service versions are invoked.

We can report that the development time for the new versions of the BPEL processes has been reduced by 18% or more for all processes that had three or more versions. For the processes with ten or more versions, the effort has been reduced by more than 27%. To compare the required development time, we have measured the time required for writing the BPEL code, the time required for integrating the BPEL with the partner link services, and the time required for deploying the new version to the process server (including the time to find the appropriate service endpoints, to register the process with the service bus, registry, etc.). We have measured the time taken by the developers before and after using *WS-BPEL Extensions for Versioning*. We have measured the total development time. To assess the actual benefit of version extensions, it would be even better to measure only the time required for version-specific activities (without the time needed for BPEL coding). Due to practical reasons, this was not possible. To normalize the results in relation to the complexity of the implemented changes, we have used the source lines-of-code metric. A better approach would be to use a more sophisticated approach, such as function points, but this was again not possible due to practical reasons. Despite this, we could demonstrate that the proposed version extensions reduced the overall development time considerably and improved the timeliness of delivery, which is an important success indicator for software projects [48]. If we would be able to measure only the time required for version-specific activities we could report even higher gains. With this, we have confirmed the real-world usability of the proposed *WS-BPEL Extensions for Versioning*.

8. Related work and discussion

Versioning of SOA business processes and BPEL versioning have not been addressed yet in a way comparable to the approach proposed in this paper. Non-existing support for versioning in SOA has been identified as an important problem [45,32]. Several authors have proposed workarounds for services in general, such as [26,18,6,36]. They have proposed to publish different versions of the same service under different endpoint URLs. This approach has often been combined with the use of naming conventions for XML namespaces. For example, we might have used namespace http://www.uni-mb.si/bookProcess/1/0, where “1” stands for major version and “0” stands for minor version. Sometimes a façade (or mediation) service has been introduced, which has decoupled the actual service implementation from the client. Façade service has been used to mask the differences between the implementation of service and different service versions.

The described approaches have numerous disadvantages: they do not address versioning of different artifacts related to BPEL process development, version traceability and efficient version management are not supported, reuse is made more difficult due to large number of endpoints and non-existing naming conventions for versions. There has also been no standard way of telling clients which is the current version of a process and which versions are deprecated. Versioning of partner links has not been supported. The described workarounds have not addressed specific needs of processes but have addressed services in general. They have not addressed the problems of long-running processes, and problems with versions when a process is upgraded while instances are running, as identified in Section 4 of this article. These workarounds therefore cannot replace the full-featured solution, as the one proposed in this article.

We have also identified some related work, which is complementary to ours. Ginige et al. [16] have addressed the problem of changes in partner services, which are consumed by BPEL processes. Such changes require alterations to BPEL processes, which is an arduous task that often leads to inconsistencies and errors. Although this observation is similar to ours, authors addressed the problem from a different perspective. They presented a way to map BPEL processes and related WSDL service descriptions into algebraic expressions to identify the effect of service changes in the orchestrated process and to implement modifications. Neth et al. [33] have described the support for versioning of business processes and human tasks in IBM WebSphere Process Server. This support is limited to the deployment-time versioning only. It does not extend BPEL and does not allow to control the partner link versions. No other authors have addressed versioning of BPEL processes, in spite of the fact that BPEL has been extended in several other directions, as described in Section 2.

Versioning and change management of business processes and workflows in general has been addressed by some authors. Dongsoo et al. [11] identified and categorized business process change patterns and proposed a new version management method for managing process changes more flexibly. Their approach is based on abstract process execution with run-time encapsulation of a business process in dynamic business environment, which is completely different from our approach. We propose extensions to BPEL for specifying version information for executable and abstract processes. Fang et al. [13] addressed three aspects of dynamic adaptation in a BPM system: model-level, instance-level, and run-time environment changes. Based on analysis, they proposed a system design for the run-time environment to support dynamic instance-level changes. In contrast, our work focuses on processes, not on process instances, and presents an integrated development, deployment and run-time approach to versioning. Tripathi and Hinkelmann [44] presented a methodology and a system for changing SOA-based business process implementations. Their approach is based on ontology-based semantic markup language for web services OWL-S. For execution, the processes are translated into BPEL. Our approach does not address implementation only, but also deployment and run-time. Our approach does not use OWL-S, but proposes extensions to BPEL directly.

Some related work can be found on versioning of web services and components in general, although authors addressed it from a different perspective than we. Kaminski et al. [23,24] addressed the problem of simultaneous deployment of multiple versions of a web service in the face of independently developed unsupervised clients. They proposed to use a form of a design technique called chain of adapters to make version-related reconfiguration tasks safe. Our approach is based on explicit version information and version-aware partner links. Ponnekanti et al. [37] studied substitution of one vendor service for another, assuming that these services are derived from a common base. Our approach is complementary to Ponnekanti et al. [37] and Kaminski et al. [23,24]. Senivongse [41] presented a model to alleviate the problem of evolving services by making different distributed service versions substitutable. He used a mediator-based approach. Cook and Dage [9] proposed a framework for upgrading components that, instead of removing old versions of components, keeps multiple versions running with the objective to improve reliability. It uses a façade approach to mask the differences between versions from the client. These two approaches differ considerably from our proposal, which relies on the assumption that the process and partner links should be version-aware. Brada [5] proposed a generic high-level representation for version history, which can be used by different component models, however author does not show how. In this part, our approach is somehow related, as we also use version history. However, we specify version history directly in the process. Our approach also addresses several other aspects, not covered in Brada [5]. Liu and Smith [31] proposed a formal labeled transition system with properties, including the fact that if a component is shipped with a certain version dependency, then at run time that dependency must be satisfied with a compatible version. This article addresses deployment time version resolution. Our approach also solves this problem, however in a different manner, as it introduces versioning as an integral part of development and run-time/deployment. There have also been some attempts to address versioning of XML schema. Snodgrass et al. [42] proposed a solution based on bundles and temporal annotations that describe changes. Our proposal extends BPEL, not schema. Our approach is based on version sequence, not on initial bundle and temporal changes. Therefore Snodgrass et al. [42] and our approach are complementary.

Versioning in general has a long history in computer science. Klahod et al. [27] proposed a general model for version management expressed by version environments. Although this approach has been focused on databases, it has some parallels with our approach. Our proposed extensions also propose version partitions, which are to a certain extent conceptually comparable to the version environments in Klahod et al. [27]. However, our approach does not address versioning of data. It provides traceability of versions and is based on service-oriented concepts. Lie et al. [29] and Reichenberger et al. [38], among others, have presented concepts and techniques for software version control. Zeller [49] presented version set model, where versions, components, and aggregates are grouped into sets according to their features. Lin and Reiss [30] presented a framework that handles versions directly in terms of the functions and classes in source code. Conradi and Westfechtel [8] has classified different versioning paradigms, and defined and related fundamental versioning concepts. Our approach builds on some of the conclusions from these articles. Above-mentioned articles however focus on development-time versioning only. At that time, run-time versioning has not been very important, because applications have been built as self-contained packages. Today in distributed, interoperable, loosely coupled environments, such as SOA, run-time versioning is increasingly important. Initial ideas of run-time versioning, at that time called instance versioning, have been mentioned by Katz [25] and Talens and Oussalah [43]. Our approach addresses development-time and run-time versioning in an integrated manner. Westfechtel et al. [46] presented a uniform version model and support architecture for software configuration management. They defined a base model that is built on a small number of concepts. Specific version models may be expressed in terms of this base model. This is conceptually similar to our approach, which also proposes an initial version, which is a base for other versions. In contrast to Westfechtel et al. [46], which addresses development-time versioning (versioning of artifacts that developers deal with), our approach provides an integrated solution. Our approach also addresses specific requirements of BPEL processes as described in Section 3.

9. Conclusion

In this article, we have proposed the *WS-BPEL Extensions for Versioning*, which represent a complete solution for versioning BPEL processes at development, deployment, and run-time. We have introduced specific new activities into BPEL, such as <*bpelx:version*>. We have extended existing activities, including partner links, *invoke*, *receive*, *import*, and *onmessage* activities, and proposed version-related extensions to BPEL variables. We have introduced a version handler, which provides control over invocations of specific process or scope versions, and allows gradual deprecation and retirement of versions, and logging of client requests.

Our approach provides process-level and scope-level versioning. It also provides means to version applications that consist of several BPEL processes. To achieve this it introduces version declarations and version bunches. A specific version can be declared as the default version, which is useful for version-unaware clients. Our approach also allows that we put temporal constraints on version validities.

With the proposed approach, we have successfully solved some major challenges in BPEL versioning, including version upgrades of processes and their management, versioning of running process instances, versioning of partner links, and versioning of long-running processes. Our solution has also solved the problem of upgrading partner link services while process instances have been running. Maintenance effort for process versioning has been considerably reduced, and version migration has been simplified. We have implemented the proposed extensions on the open-source Apache ODE engine, and tested the solution in real-world environment, where we have achieved positive results.

References

[1] A. Agrawal et al., WS-BPEL Extension for People (BPEL4People), Version 1.0, 2007. <http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/BPEL4People_v1.pdf>.

[2] A. Agrawal et al., Web Services Human Task (WS-HumanTask), Version 1.0. <http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel4people/WS-HumanTask_v1.pdf>.

[3] Apache Software Foundation, 2008. Apache ODE, 2007. <http://ode.apache.org/>.

[4] S. Biffl, A. Aurum, B. Boehm, H. Erdogmus, P. Grunbacher, Value-based Software Engineering, Springer Verlag, Berlin, Heidelberg, 2005.

[5] P. Brada, Component revision identification based on IDL/ADL component specification, ACM SIGSOFT Software Engineering Notes 26 (5) (2001) 297–298.

[6] K. Brown, M. Ellis, Best Practices for Web Services Versioning, IBM developerWorks, 2004. <http://www-128.ibm.com/developerworks/webservices/library/ws-version/>.

[7] A. Charfi, M. Mezini, Aspect-oriented web service composition with AO4BPEL, in: ECOWS, LNCS, vol. 3250, Springer, 2004, pp. 168–182.

[8] R. Conradi, B. Westfechtel, Version models for software configuration management, ACM Computing Surveys 30 (2) (1998) 232–282.

[9] J.E. Cook, J.A. Dage, Highly reliable upgrading of components, in: Proceedings of the 21st International Conference on Software Engineering, 1999, pp. 203–212.

[10] G. Decker, O. Kopp, F. Leymann, M. Weske, BPEL4Chor: extending BPEL for modeling choreographies, ICWS, in: IEEE International Conference on Web Services (ICWS 2007), 2007, pp. 296–303.

[11] K. Dongsoo, K. Minsoo, K. Hoontae, Dynamic business process management based on process change patterns, ICCIT, in: International Conference on Convergence Information Technology, 2007, pp. 1154–1161.

[12] A. Erradi, P. Maheshwari, AdaptiveBPEL: a policy-driven middleware for flexible web services compositions, in: Proceedings of Middleware for Web Services (MWS), 2005.

[13] R. Fang, Z. Zou, C. Stratan, L. Fong, D. Marston, L. Lam, D. Frank, Dynamic support for BPEL process instance adaptation, SCC, IEEE International Conference on Services Computing 1 (2008) 327–334.

[14] H. Gaur, M. Zirn, BPEL Cookbook: Best Practices for SOA-Based Integration and Composite Applications Development, Packt Publishing, Birmingham, 2007.

[15] C. Ghedira, H. Mezni, Through personalized web service composition specification: from BPEL to C-BPEL, Electronic Notes in Theoretical Computer Science 146 (2006) 117–132.

[16] J. Ginige, U. Sirinivasan, A. Ginige, A mechanism for efficient management of changes in BPEL based business processes: an algebraic methodology, ICEBE, in: IEEE International Conference on e-Business Engineering, 2006, pp. 171–178.

[17] Institute for Software Research, xADL 2.0, Highly-Extensible Architecture Description Language for Software and Systems, University of California, Irvine, 2005. <http://www.isr.uci.edu/projects/xarchuci/index.html>.

[18] R. Irani, Versioning of Web Services, Web Services Architect, 2004. <http://www.webservicesarchitect.com/content/articles/irani04.asp>.

[19] M.B. Juric, R. Loganathan, P. Sarang, F. Jennings, SOA Approach to Integration, Packt Publishing, Birmingham, 2007.

[20] M.B. Juric, B. Mathew, P. Sarang, Business Process Execution Language for Web Services, second ed., Packt Publishing, Birmingham, 2006.

[21] M.B. Juric, K. Pant, Business Process Driven SOA Using BPMN and BPEL, Packt Publishing, Birmingham, 2008.

[22] M.B. Juric, A. Sasa, B. Brumen, I. Rozman, WSDL and UDDI Extensions for Version Support in Web Services, Journal of Systems and Software, 2009, doi:10.1016/j.jss.2009.03.001.

[23] P. Kaminski, M. Litoiu, H. Muller, A design technique for evolving web services, in: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research, 2006, p. 23.

[24] P. Kaminski, H. Muller, M. Litoiu, A design for adaptive web service evolution, in: Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems, 2006, pp. 86–92.

[25] R.H. Katz, Toward a unified framework for version modeling in engineering databases, ACM Computing Surveys (1990) 375–408.

[26] J. Kenyon, Web Service Versioning and Deprecation, Web Services Journal, 2003. <http://webservices.sys-con.com/read/39678.htm>.

[27] P. Klahod, G. Schlageter, W. Wilker, A general model for version management in databases, in: Proceedings of the Twelfth International Conference on Very Large Databases, 1986, p. 319.

[28] M. Kloppmann et al., WS-BPEL Extension for Sub-Processes – BPEL-SPE, A Joint White Paper by IBM and SAP, September 2005, 2005. <http://download.boulder.ibm.com/ibmdl/pub/software/dw/webservices/ws-bpelsubproc/ws-bpelsubproc.pdf>.

[29] A. Lie, T. Didriksen, R. Conradi, E. Karlsson, S. Hallsteinsen, P. Holager, Change oriented versioning, Lecture Notes in Computer Science 387 (1989) 191–202.

[30] Y. Lin, S.P. Reiss, Configuration management with logical structures, in: 18th International Conference on Software Engineering (ICSE'96), 1996, p. 298.

[31] D.Y. Liu, S.F. Smith, A formal framework for component deployment, ACM SIGPLAN Notices 41–10 (2006) 325–344.

[32] P. Louridas, Orchestrating Web Services with BPEL, IEEE Software 25 (2) (2008) 85–87.

[33] J. Neth, M. Smolny, C. Zentner, Versioning Business Processes and Human Tasks in WebSphere Process Server, IBM developerWorks, 2008. <http://www.ibm.com/developerworks/websphere/library/techarticles/0808_smolny/0808_smolny.html>.

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

[35] Oracle, Oracle BPEL Process Manager Developer's Guide 10 g (10.1.3.1.0), Part Number B28981-03, 2007. <http://download.oracle.com/docs/cd/B31017_01/integrate.1013/b28981/toc.htm>.

[36] C. Peltz, A. Subbarao, Design Strategies for Web Services Versioning, Web Services Journal, 2004. <http://webservices.sys-con.com/read/44356.htm>.

[37] S.R. Ponnekanti, A. Fox, Interoperability among independently evolving web services, in: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware, 2004, pp. 331–351.

[38] C. Reichenberger, Concepts and techniques for software version control, Software – Concepts and Tools 15 (3) (1994) 97–104.

[39] G. Ruhe, M.O. Saliu, The art and science of software release planning, IEEE Software (2005) 47–53.

[40] P. Rovegard, L. Angelis, C. Wohlin, An empirical study on views of importance of change impact analysis issues, IEEE Transactions on Software Engineering 34 (4) (2008) 513–530.

[41] T. Senivongse, Enabling flexible cross-version interoperability for distributed services, in: Proceedings of the International Symposium on Distributed Objects and Applications, 1999, p. 201.

[42] R.T. Snodgrass, C. Dyreson, F. Currim, S. Currim, S. Joshi, Validating quicksand: temporal schema versioning in tXSchema, Data and Knowledge Engineering 65 (2) (2008) 223–242.

[43] C. Talens, C. Oussalah, M.F. Colinas, Versions of simple and composite objects, in: Proceedings of the 19th VLDB Conference, Ireland, 1993, pp. 62–72.

[44] F. Tripathi, K., Hinkelmann, Change Management in Semantic Business Processes Modeling, ISADS, in: Eighth International Symposium on Autonomous Decentralized Systems (ISADS'07), 2007, pp. 155–162.

[45] S. Vinoski, The more things change, IEEE Internet Computing 8 (1) (2004) 87–89.

[46] B. Westfechtel, B.P. Munch, R. Conradi, A layered architecture for uniform version management, IEEE Transactions on Software Engineering 27 (12) (2001) 1111–1133.

[47] Wikipedia, Software Versioning, 7 October 2008. <http://en.wikipedia.org/wiki/Software_versioning>.

[48] C. Wohlin, A. von Mayrhauser, M. Höst, B. Regnell, Subjective evaluation as a tool for learning from software project success, Information and Software Technology 42 (14) (2000) 983–992.

[49] A. Zeller, A Unified Version Model for Configuration Management, in: Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, 1995, pp. 151–160.