A Classification of BPEL Extensions

Oliver Kopp[1], Katharina Görlach[1], Dimka Karastoyanova[1], Frank Leymann[1], Michael Reiter[1], David Schumm[1], Mirko Sonntag[1], Steve Strauch[1], Tobias Unger[1], Matthias Wieland[1], Rania Khalaf[2]

[1]Institute of Architecture of Application Systems, University of Stuttgart, Germany
{wieland, goerlach, schumm, leymann}@iaas.uni-stuttgart.de

[2]IBM TJ Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
rkhalaf@us.ibm.com

BibTeX:

```
@article{KoppGK+2011,
  author    = {Oliver Kopp and Katharina G\"{o}rlach and Dimka Karastoyanova
               and Frank Leymann and Michael Reiter and David Schumm and
               Mirko Sonntag and Steve Strauch and Tobias Unger and
               Matthias Wieland and Rania Khalaf},
  title     = {A Classification of {BPEL} Extensions},
  journal   = {Journal of Systems Integration},
  year      = {2011},
  volume    = {4},
  number    = {2},
  pages     = {3--28},
  issn      = {1804-2724},
  publisher = {Czech Society for Systems Integration (CSSI)}
}
```

ISSN: 1804-2724

A Classification of BPEL Extensions

Oliver Kopp, Katharina Görlach, Dimka Karastoyanova, Frank Leymann, Michael Reiter, David Schumm, Mirko Sonntag, Steve Strauch, Tobias Unger, Matthias Wieland
Institute of Architecture of Application Systems, University of Stuttgart
{lastname}@iaas.uni-stuttgart.de

Rania Khalaf
IBM TJ Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, USA
rkhalaf@us.ibm.com

Abstract: *The Business Process Execution Language (BPEL) has emerged as de-facto standard for business processes implementation. This language is designed to be extensible for including additional valuable features in a standardized manner. There are a number of BPEL extensions available. They are, however, neither classified nor evaluated with respect to their compliance to the BPEL standard. This article fills this gap by providing a framework for classifying BPEL extensions, a classification of existing extensions, and a guideline for designing BPEL extensions.*

Key words: BPEL Extension, Classification of Extensions, Extension Guidelines

1. Introduction

Originally, the Business Process Execution Language (BPEL) has been designed for the implementation of business processes using Web service technology. The Web service technology is the de-facto standard used to implement a service-oriented architecture (SOA, Weerawarana et al., 2005). Nowadays, BPEL is used for implementing business processes in numerous different scenarios: for automating scientific simulations, for provisioning software as a service (SaaS) applications and as exchange format for business processes (i.e., BPEL as description language for business protocols). The requirements of the usage scenarios differ and the desired functionality is not always shipped out of the box, i.e., it is not supported using standard language constructs. For instance, sub-processes are a demand that the BPEL specification (OASIS, 2007) and consequently standard-conform implementations do not cover. As a result, BPEL is frequently extended for supporting desired functionality that is not available in standard BPEL. Depending on the particular purpose, an extension may improve efficiency, increase flexibility, ensure better performance, or add more functionality. However, an extension also has disadvantages. Firstly, the whole toolset that is used for business process management (BPM) needs to support the extension. Common components of this toolset are applications for modeling, adapting, executing, monitoring, and analyzing the processes. Secondly, if business partners exchange (parts of) their processes, their toolsets need to understand and support the extensions as well.

In this paper, we provide a classification of existing BPEL extensions and provide guidelines to develop extensions. This might support a developer to search for existing extensions and to develop a new extension in case a new one is necessary. Consequently, the paper is structured as follows: Sect. 2 provides the technical background that describes the typical environment for BPEL processes as well as the associated components and technologies. Sect. 3 introduces a classification framework for extensions including standard-conformity, distinction between modeling and runtime extensions, as well as different purposes. Building on this, Sect. 4 presents requirements on extensions to be standard-conformant to BPEL. Sect. 5 presents approaches to realize a BPEL extension and the related BPEL environment. Sect. 6 introduces an extension development guideline that helps in the course of implementing an extension. The classification is applied to 62 existing BPEL extensions in Sect. 7. The paper finishes with a conclusion in Sect. 6.

2. Background

In the following we describe the environment that is common for using workflows (cf. Fig. 1). Workflows are the implementation of business processes (Leymann & Roller, 2005). The environment also applies to environments for BPEL processes.

The components in the upper part of the figure represent the modeling part of the environment. It consists of three components. The process modeling tool is used for the (typically graphical) specification of process models. The process analysis tool refers to static verification, deadlock analysis and other checks that can be performed at design time. It is often already integrated in the process modeling tool. Finally, the process repository serves as a means for efficient storage and retrieval of process models.

The components in the lower part of Fig. 1 represent the runtime environment. The central component for runtime is the process engine. At process deployment time, a process model is passed to this component, which compiles the process model into an internal format and offers the deployed process as a service to the outside. A so-called navigator, a subcomponent of the process engine, manages the status of process instances, traverses workflow graphs, triggers activity implementation execution, and takes care of directing incoming messages to the intended recipients, i.e., to particular process instances using correlation (Barros et al., 2007). The process engine communicates with services via the enterprise service bus (ESB; Chappell, 2004). The ESB allows for abstracting from communication details, such as the used transport protocol and message format. Note that an ESB is an abstract concept which may be implemented using a specific component (which is generally referred to as ESB, too) or in other ways, such as embedded into the process engine (cf. Leymann, 2005). The services represent the actual functions that are orchestrated in the workflow. The monitoring component registers, receives, and analyses execution events that are emitted by the process engine and the orchestrated services. For example, this component allows tracking the status of a particular instance of a process.

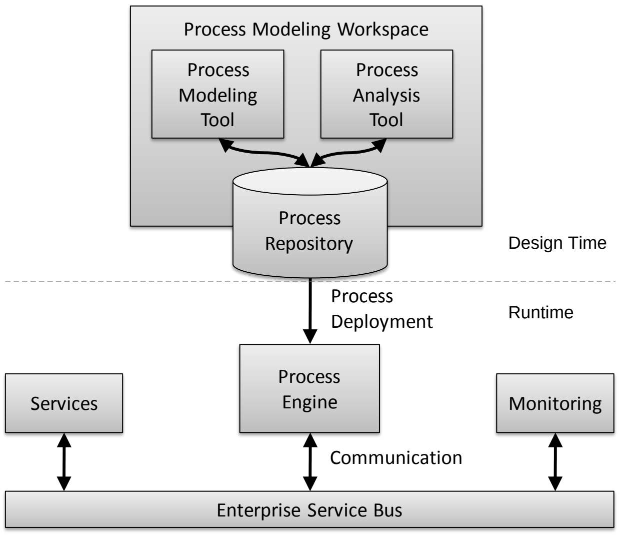

Fig. 1: Common Environment for Workflows

BPEL is a workflow language for specifying business process behavior based on Web Services (OASIS, 2007). It provides activities to exchange messages with Web Services and provides control-flow constructs to order these activities. BPEL requires the interfaces to be specified in WSDL 1.1 (Christensen et al., 2001). It is important that WSDL does not require the messages being exchanged using SOAP/http. Other bindings, such as SOAP over Java Messaging Service are available, too (Adams et al, 2010). In BPEL, the connection to partner services is formed by a partner link, which specifies the port type required, offered, or both. An `invoke` activity is used to send a message to a specific operation of a Web Service. In its two-way form, it awaits a reply message back. A `receive` activity is used to receive a message by a given operation. A `pick` activity realizes a one-out-of-many choice of mutual exclusive incoming messages. A `wait` activity waits for a specified time or until a given date is reached. An `empty` activity does nothing. The `scope` activity enables fault-correcting behavior and event-handling. Faults are catched by fault handlers. A completed scope may be compensated. The compensation behavior is specified by a compensation handler. Event-handlers run in parallel to the activities in the scope and handle additional incoming messages and timeouts. The control-flow itself may be specified using block-structured and graph-based construct, which makes BPEL a hybrid workflow language (Kopp et al., 2009). The block-based constructs are `sequence`, `if`, `while`, `repeatUntil`, `forEach`, and `flow` without links. A `flow` with links enables modeling of a graph, where control-flow follows the specified links. A detailed summary is provided .by Leymann & Roller (2006).

The first version of BPEL has been proposed in 2002 as "Business Process Execution Language for Web Services 1.0" (BPEL4WS). Subsequently, version 1.1 has been released in 2003. Here, minor corrections and clarifications have been made. This version has been submitted to the Organization for the Advancement of Structured Information Standards (OASIS). In 2007, OASIS has completed the standardization process and has published the revised version as *WS-BPEL 2.0*. Important changes have been made with respect to the extensibility of the language. For example, designated concepts such as an `extensionActivity` element or `extensionAttributes` have been added (cf. Sect. 4). A detailed comparison of the BPEL versions 1.1 and 2.0 is provided by Schumm (2007). The work we present in the following focuses on the current language specification WS-BPEL 2.0 (BPEL), and the extensibility mechanisms specified therein. Where appropriate, we point out properties of BPEL 1.1.

In order to refer to the components affected by an extension, we present an exemplary architecture of a BPEL engine. We implemented a prototype of a BPEL engine (called Stuttgart's Workflow Machine, SWoM) at our institute[1]. The architecture of SWoM distinguishes all major components existing in a BPEL engine and thus can be used to illustrate them. The internal architecture of the SWoM is illustrated in Fig. 2. It consists of four main modules namely Gateway, Process Execution, Persistence, and Administration. The Gateway deals with Web Service invocations and handles incoming messages. The Process Execution is responsible for process instance creation and execution. The Persistence consists of databases for storing auditing events (*Audit Database*), data about deployed BPEL process models with appropriate WSDLs and deployment descriptors (*Buildtime Database*), and information about process instances (*Runtime Database*). The Administration contains an interface and functionality for human users to supervise process execution. The arrows in the figure indicate communication dependencies. Message queues and topics are used to decouple modules. Components with a black box at the top expose their functionality as Web service. After giving a short overview regarding the main modules in the following, we describe their inner structure.

The *Administration Interface* enables human access to core functionality of the engine. The *Import Export Handler* is used to import process models into the engine, to statically validate process models, and to delete and export uploaded process models. The *Process Deployment Manager* is responsible for deployment and undeployment of imported process models. With the help of the *Supervision* an administrator can activate or deactivate the auditing of process models. Furthermore, audited events of process models can be inspected. The *Systems Management* allows viewing and deleting errors occurred in the SWoM, forced termination of running process instances and their deletion from the SWoM as well as user management. The *Administration Infrastructure Provider* is an interface to access the databases and to put messages into the Manager topic (indicated by an "MT").

The *Service Provider* component exposes deployed process models as Web services. Web service clients can invoke processes by sending a SOAP message to the engine. In case of a synchronous request/response operation the Service Provider additionally sends the reply back to the client. The *Invocation Handler* is responsible for the invocation of Web services following the blocking request/response pattern or the unblocking one-way pattern.

The *Navigator* interprets process model logic, supervises control and data flow, and executes activity implementations. It makes use of the navigation queue (indicated by an "N") to send and receive navigation events. For each `invoke` activity it puts a Web service invocation message into the invocation queue (indicated by an "I") to be performed by the Invocation Handler. In case of a reply activity it inserts a reply message into the reply queue (indicated by an "R") to be sent back to the invoking client by the Service Provider. The *Data Manager* provides Runtime database access to the Navigator and caches process models to prevent from extensive Buildtime database accesses during process execution. Steering of *Data Managers* can be done over the Manager topic, e.g., to force process model state changes. The *Auditing* persistently stores information about the life of a process for analysis or legal reasons. The *Process Instance Creator* is used by the Navigator to build new

[1] Institute of Architecture of Application Systems (IAAS), http://www.iaas.uni-stuttgart.de/institut/

process model instances in the Runtime database. The *Correlation Manager* correlates incoming and outgoing messages to corresponding process instances.

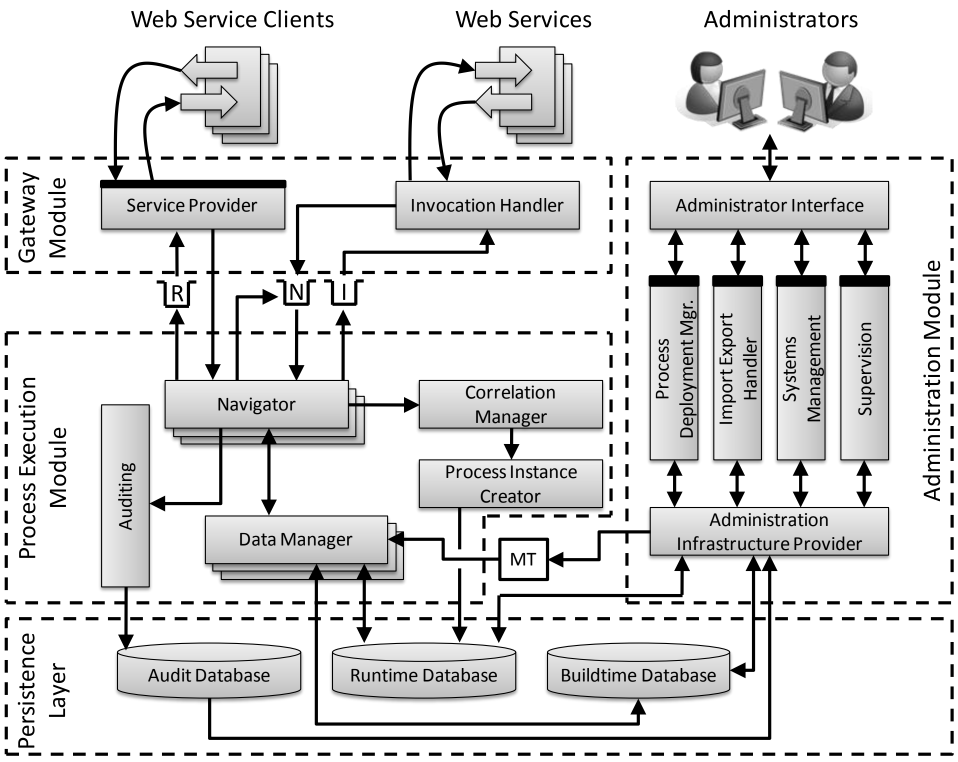

Fig. 2: Architecture of the BPEL Engine SWoM

Extending a process language has profound impact on all components of its supporting infrastructure, most important on the modeling tool and the process engine. Furthermore, the other components involved, such as tools for process analysis and monitoring, have to be adapted accordingly. Our evaluation of current approaches for extending BPEL in Sect. 7 shows that most extensions cover modeling tool and runtime extensions only.

3. Classification Framework

The follwoing definition defines the term “BPEL extension” and is referred to throughout the paper. The definition follows the definition of a software extension in the field of computer science (Laemmel and Ostermann, 2006).

> ***Definition 1:*** *A standard-conform BPEL extension is an enhancement of functionality of the Web Services Business Process Execution Language specified in the OASIS WSBPEL 2.0 standard by following the extension proceedings defined in the standard. On its own, the BPEL extension is not useful or functional.*

To be standard-conformant, extensions must not contradict the semantics of any element or attribute defined by the WS-BPEL specification. The concrete guidelines defined in the WS-BPEL 2.0 standard (OASIS, 2007) are summarized in Sect. 4. The essence of these guidelines is presented in Tab. 1. In this table we provide a checklist for classifying a given extension with respect to its standard conformity. The table shows a characteristic, its standard conformity, and an identifier as a shortcut. The shortcut is used in Sect. 7 as reference for a classification. BPEL 1.1 does not explicitly define an extension mechanism, but allows for adding elements of other namespaces into the process model. BPEL 2.0 explicitly specifies the extension mechanism of BPEL. This has impact on the standard-conformity of an extension. As a consequence, we show the BPEL version in the column “Standard-conform Language Extension”. In case several characteristics are applicable to an extension, an extension has to be standard-conformant regarding all characteristics. Tab. 2 provides a classification into design time and runtime extensions. The runtime components listed in Tab. 2 are components illustrated in Fig. 2 which were extended by the extensions presented in Sect. 4. Note that an extension can be both a design time extension and a runtime extension. “n/a” denotes “not applicable”. This is the case if an extension is not a BPEL extension the sense of Definition 1. For instance, in case an extension changes the behavior of an invocation handler only, it is not an extension in the sense of Definition 1. For a standard-conform runtime extension at least the navigator has to be extended.

Tab. 1: Standard Conformity

Characteristic	**Standard-conformant Language Extension**	**Shortcut**
New activity without nesting in an extension activity	No (2.0) / Yes (1.1)	$\overline{s}$ 1
New construct/element in BPEL namespace	No (1.1/2.0)	$\overline{s}$ 2
New attribute in BPEL namespace	No (1.1/2.0)	$\overline{s}$ 3
Contradiction with BPEL semantics	No (1.1/2.0)	$\overline{s}$ 4
Defining something out of scope of the BPEL specification (not using <process> as root element, …)	No (1.1/2.0)	$\overline{s}$ 5
No extension declaration specified	No (2.0) / Yes (1.1)	$\overline{s}$ 6
New extension activity	Yes (2.0)	s7
New extension attribute	Yes (1.1/2.0)	s8
New extension construct/element	Yes (2.0)	s9
New extension assign operation	Yes (2.0)	s10

Tab. 2: Extension Type

Extension	**Type**	**Shortcut**
Modeling tool extension		
BPEL Extension can be transformed to standard BPEL	Modeling	M
BPEL Extension cannot be transformed to standard BPEL	Modeling	M
Modeling tool offers different rendering	n/a	
Process engine extension		
Deployment mechanism extension	n/a	
Invocation handler extension	n/a	
Correlation manager extension	n/a	
Navigator extension	Runtime	R
Auditing extension	n/a	

Extensions can be further characterized, independent of their standard-conformity and particular type. We use the extension purpose, the extension subject, the workflow dimension, and the placement in the business process management (BPM) life cycle as additional characterizations. The *extension purpose* criterion lists different intentions of an extension, such as the improvement of reusability of processes. The *extension subject* addresses the language constructs and mechanisms which are affected by an extension. According to Leymann and Roller (2000) a workflow has three independent dimensions (IT infrastructure, process logic, and organization). We use these *workflow dimensions* as one criterion to characterize an extension. Finally, we use the *placement in the BPM life cycle* as criterion. The life cycle starts with modeling a business process. This business process has then to be refined to an executable process model (IT refinement). Static analysis and verification makes sure that the process model conforms to given constraints (e.g., freeness of deadlocks). Subsequently, the process model is deployed on a process engine, where the process is executed. In the monitoring phase the execution of single processes or process groups is observed. The results of monitoring are analyzed and may lead to redesign and optimization, which is again conducted in the modeling phase closing the loop.

These extension characteristics are listed in Tab. 3. We have derived the criteria and appropriate characteristics from the evaluated extensions (see Sect. 4). This list may be further extended when discussing novel extensions. The characteristics are sorted alphabetically, except the life cycle characteristics, which are sorted according to the order in the life cycle. “Occurrence” shows the total number of extensions matching the respective characteristic

Tab. 3: Extension characteristics

Criterion	**Characteristic**	**Shortcut**	**Occurrence**
Purpose	Ability to outsource	C1.1	3
	Flexibility	C1.2	13
	Functionality	C1.3	28
	Maintainability	C1.4	13
	Performance	C1.5	6
	Reusability	C1.6	9
	Robustness	C1.7	11
	Usability	C1.8	12
Subject	Control flow	C2.1	25
	Data integration	C2.2	10
	Expressions/assign statements	C2.3	3
	Handling of large data	C2.4	2
	Other	C2.5	9
	Service binding	C2.6	5
	Service invocation	C2.7	22
	Variable access	C2.8	3
Workflow dimension	IT infrastructure	C3.1	29
	Process logic	C3.2	36
	Organization	C3.3	2
Placement in the BPM life cycle	Modeling	C4.1	47
	IT refinement	C4.2	2
	Static analysis/verification	C4.3	0
	Deployment	C4.4	10
	Execution	C4.5	45
	Monitoring	C4.6	3

Based on Definition 1, we can *exclude* particular changes on the BPEL language and give a list of approaches, which are *not* a BPEL extension. BPEL offers the possibility to model abstract processes, which need not to be executable but address different use cases. An abstract process profile specifies the semantics of an abstract process. It furthermore describes how to get an executable process starting from the abstract one, called “executable completion”. The BPEL specification itself provides two profiles: A profile for observable behavior and a profile for process templates. Abstract processes following the abstract process profile for observable behavior describe the public visible behavior of a process. Abstract processes following the template profile serve as process templates, where activities required for execution have to be put in at fixed places. König et al. (2008) introduce the Abstract Process Profile for Globally Observable Behavior, which enhances the profile for observable behavior by providing more flexibility for the executable completion. Describing a new Abstract BPEL process profile is not an extension as it is just a restriction that defines, which constructs are allowed in a process model.

Approaches that redefine the semantics of existing BPEL constructs are not standard-conform and thus not an extension in the meaning of Definition 1. The specification does not provide information about the event model a process engine should support. Hence, a modification or extension of an existing event model, such as defined by Karastoyanova et al. (2006), is out of scope of the specification and thus not a BPEL extension.

BPEL itself does not specify any rendering of the process model. Since the rendering is not standardized, any specific rendering is not a BPEL extension. This includes graphical renderings in BPMN (Schumm et al., 2009; Weidlich et al., 2008) or a script syntax such as BPELscript (Bischof et al., 2009).

4. Requirements for Standard-conform Extensions

In BPEL 2.0, the extensibility of BPEL is standardized. Extensions are declared in the `extensions` element. Each extension is associated with a namespace and takes a Boolean attribute `mustUnderstand` (OASIS, 2007, Sect. 14). In case the value is set to "`yes`", a process engine has to reject the process model if it does not support the extension. The specification does not state anything about the modeling tool. A value of "`no`" denotes that the extension is optional. In case an engine is not aware of the extension, the each respective `extensionActivity` is replaced by an `empty` activity, extension assignments are ignored, and all other XML attributes and XML elements are ignored.

The BPEL standard offers following possibilities to extend the language:

- Introduce new activity types, called `extensionActivity` (OASIS, 2007, Sect. 10.9)
- Include new data manipulation operations (OASIS, 2007, Sect. 8.4)
- Specify individual query and expression languages (OASIS, 2007, Sect. 8.2)
- Allow namespace-qualified attributes and elements from other namespaces (OASIS, 2007, Sect. 5.3) and apply extension semantics for all BPEL constructs in the syntax sub-tree (OASIS, 2007, Chapter 14)

The standard requires that an extension does not cause any change to the semantics of a BPEL process (OASIS, 2007, Sect. 5.3). If an `extensionActivity` is a start activity or contains a start activity, the namespace of the `extensionActivity` child element must be declared as `mustUnderstand="yes"` (OASIS, 2007, Sect. 10.4). In the old version of BPEL, namely BPEL 1.1, an extension is simply made by adding XML attributes and XML elements in another namespace into the BPEL process. In case a workflow engine is not aware of the namespace, the behavior is not specified by the BPEL 1.1 specification. This version of the specification does not impose any restrictions on extensions. The fact that the execution semantics of the extension has to be described is implicitly required by all versions of the specification.

5. Possibilities to Realize an Extension

We distinguish between two different options for the realization of an extension in terms of Definition 1: (A1) Extended modeling tool and extended engine and (A2) extended modeling tool and model transformation.

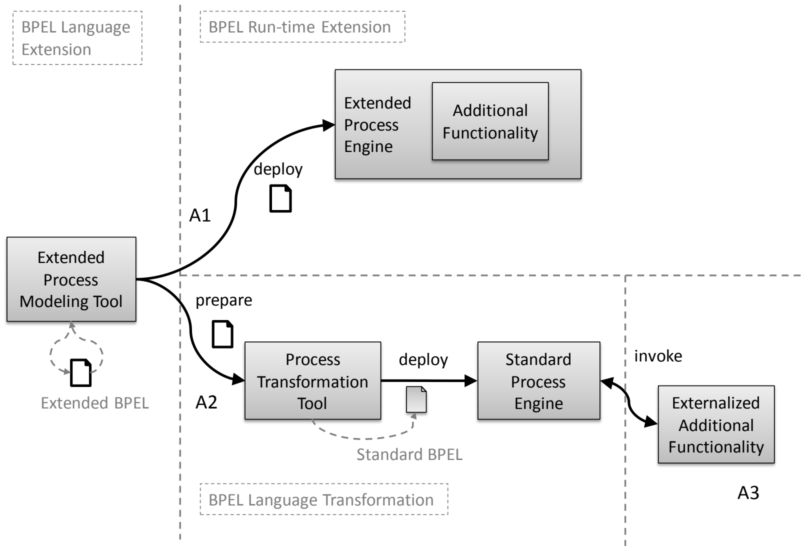

Fig. 3: Runtime Extension versus Model Transformation

The first option A1 "BPEL Runtime Extension" is represented by the upper branch in Fig. 3. Extended BPEL code is created in a modeling tool which supports this kind of extension. The BPEL code and its extension are deployed onto a process engine that supports the additional functionality. That means, the process engine has to be modified for this option.

The second option A2 "BPEL Language Transformation" is represented by the lower branch in Fig. 3. Extended BPEL code is created in an extended modeling tool as well. The significant difference to A1 is the employment of model transformations (Stahl et al., 2006). This technique translates the extension constructs into standard BPEL language constructs. Standard BPEL code is thereby generated that can be deployed on a process engine that is not aware of any extension. Note that this paper does not discuss transformations of other business process modeling languages to BPEL. A discussion of that aspect is given by Stein et al. (2009).

In addition to these two options, there is the possibility to separate the desired functionality in an external service. In case an extension uses this approach, it is not a valid extension according to Definition 1. This option A3 "Dedicated Service" is shown in the lower right corner in Fig. 3. The external service can be invoked from the process engine with standard language constructs. That means, a language extension is not required per se, but typically provides more comfort. In this setting, the modeling tool may be extended to support different renderings of the dedicated services or may be kept as is.

The runtime extension approach (A1) envisages extending both the language (including the modeling tool) and the execution engine that supports the execution of the new constructs. This may also require an adaption of the monitoring components, as they may need to distinguish standard and extended activities and monitor them differently. The consequential changes may reach up to the dashboard. A prominent example for the runtime approach is the extension BPELJ (Blow et al., 2004), which extends BPEL with the possibility to use Java code snippets as an activity. The BPEL language is extended with an according `extensionActivity`, the modeling tool is extended for support of entering Java code, and also the process engine is extended for actually executing the Java code.

In the model transformation approach (A2) basically higher level constructs are introduced. This is, however, only possible if an extension is expressible with a set of standard constructs. For illustrating this approach we take a fictive BPEL extension, which we call "Delayed Execution". Listing 1 shows the code for an `invoke` activity that uses the "Delayed Execution", which delays the execution for 3 days and 10 hours counted from the point of the activation of the `invoke`.

```
<invoke name="refreshValue" ext:delay="P3DT10H" .../>
```

Listing 1: Invoke Activity Extended for Delayed Execution

A model transformation tool has to processes all constructs that carry an extension attribute for the delay. Each identified construct is split up into a `wait` activity and the actual activity (here: an `invoke` activity) that should be executed (cf. Listing 2).

```
<sequence …>
  <wait name="refreshValueDelay" for="P3DT10H" />
  <invoke name="refreshValue" …/>
</sequence>
```

Listing 2: Extended Invoke Activity Transformed to Standard Constructs

For some cases, functionality can be externalized as a service (A3). This approach is easy to implement, offers high reusability (even outside of BPEL processes), and does not hamper portability of the processes. A major issue is that the require functionality may need the current state of a process instance such as the state of activities and variable content, which is difficult to pass to the externalized service. This limits the applicability of this approach. The approach may, for instance, be applied for extending BPEL with business rules, discussed in Sect. 7.2.1.

When comparing the different extension options A1 and A2, the model transformation approach (A2) has one significant advantage: Compatibility and thus portability of the process models to another toolset is preserved. It also has a significant disadvantage: The original activity is replaced by a set of new activities, variable definitions, and other constructs that do not represent the work that was actually intended. This circumstance impacts monitoring and debugging instruments that will register the execution of activities that are not contained in the original process model. To ease monitoring, an additional transformation step of monitoring information into the former process model format is required. The transformation approach is, however, not applicable in all cases. If an extension cannot be expressed with standard constructs, an extension of the engine is inevitable. The advantage of the runtime extension approach is the holistic and consistent integration of the extension in the modeling tool and workflow engine. The user gets what he modeled. Moreover, this solution promises the highest engine performance due to an optimized workflow model (as no additional elements are generated) and a reduced communication overhead. The disadvantage of the runtime extension approach is that it requires a huge development effort. Note that the approaches can be combined: In case a process engine supports the extension, it can be executed natively. If it does not support the extension, then a model transformation step needs to take place in advance. The option to externalize the new functionality into a distinct service that offers the functionality is only possible if the new functionality does not affect actual engine components, such as the navigator.

6. Extension Development Guideline

If new functionality is required for the development of business processes, one has to balance how and where to integrate this functionality. This section provides the reader a means at hand to decide whether a BPEL extension is an adequate solution. For supporting the decision making, we present in Sect. 6.1 different aspects that should be considered when planning BPEL extensions and give recommendations how to achieve the planned goal. Due to its high development effort, the runtime extension approach (A1) should be avoided if possible. If no reason is found for an A1 extension, the enhanced functionality should be implemented in other ways as described in Sect. 5. For instance, the enhanced functionality may be realized as Web service called by a workflow, as functionality in the ESB-infrastructure, as design time extension in the modeling tool, or as transformation. If it turns out that the A1 approach is needed, there are different possibilities how to implement it. Sect. 6.2 discusses three possibilities: As a commercial solution, as a self-implemented solution based on open source software, or as a hosted solution.

6.1. Recommendations for the Choice of Extensions

When deciding about the need for an extension, different aspects of the extension have to be thought of, which we present in the following. We discuss the aspects and give recommendations for design and implementation of extensions. Note that the considered aspects are arranged in an unordered list.

Implementation of the functionality in other components of the infrastructure The infrastructure offers components such as an enterprise service bus or application server (cf. Sect. 2). It may be possible to implement the needed functionality in a component other than the engine. For example, retrying service invocation or replacing a service with an equivalent service is a typical task for an ESB (Chappell, 2004; Leymann, 2005). Thus, this functionality is not implemented in the BPEL engine but in the integration layer. If the functionality can be realized by modifying infrastructure components other than the engine (e.g., the ESB), we recommend this approach. In case the planned extension needs to be reflected in the workflow logic, it should be implemented in the workflow engine.

Visibility of the extension in the workflow model required A BPEL extension is visible in the workflow model if it is explicitly declared as `extension` and either embedded in an `extensionActivity/extensionAssignOperation` element or implemented as an extension attribute or extension element. This allows identifying usage of the extension easily. In case visibility of the extension is necessary for process users and/or developers, the extension should be designed according to the standard mechanism (cf. Sect. 5) using the A1 or A2 approach. If visibility can be neglected, we recommend the dedicated service solution (A3), which is easier to implement.

Visibility of the extension in the audit trail required Typically, a BPEL engine logs state changes of activities in the audit trail. The planned extension may need to be accounted for in the audit trail. When realizing the extension with approach A2, the process model is transformed into a standard BPEL process model where the extension is not visible anymore. We recommend solving this problem with a two-directional mapping between the modeling tool extension and the representing standard BPEL elements. The mapping can be used to conciliate the displayed auditing information and the process model. It may happen that the backward mapping (transformed model to extended design time model) is complex or even ambiguous, e.g., in the case a service is used by multiple extensions. In this case we recommend realizing the extension in the workflow engine (A1).

Detailed internal execution information of the extension in the audit trail required It may be required to add information beyond standard state changes of an extension activity to the audit trail, which may be the progress of execution or the selected user for instance. In this case an engine extension is inevitable (A1). Furthermore, the audit trail has to be capable of storing this additional information and may also need to be extended.

Execution performance importance If the runtime of the extended functionality is a major issue, we recommend implementing the extension directly in the workflow engine (A1). This solution is characterized by the possibility of optimized code (compared to A2) and by a reduced communication overhead (compared to A3).

Based on the decision taken at each aspect, it can be decided whether a BPEL extension is needed. The decision depends on the concrete problem statement. Thus, a general answer cannot be given and has to be made on a per-case basis. The different alternatives, their advantages and shortcomings are presented in Sect. 5. In case the decision is to create a BPEL extension, the next step is to decide how to realize the extension.

6.2. Solution Possibilities for Implementing a BPEL Extension in a BPEL Engine

After deciding for realizing an extension in the modeling tool and the engine (approach A1 from Sect.5, the extension has to be implemented both in the modeling tool and in the engine. In case the approaches A2 or A3, the modeling tool has to be changed to support the extension. The BPEL engine stays unchanged.

In this section, we describe how extensions can be added to existing systems by providing concrete examples. The discussion is structured around several key considerations: The level of extension support in the system and the ability to modify the system itself. Subsequently, we address the additional issues arising from implementing extensions in hosted BPM systems, e.g., “BPM as a service”, which is an emerging trend.

The first consideration is whether the system (modeling tool and engine) has some or full support for extensibility. In the case that it does have support, the developer simply uses the extension support – provided that it can handle the requirements of the target extension. Examples for this case are the Eclipse BPEL designer[2] and the Apache ODE engine, where plug points for extensions are available. The “Pluggable Framework for Enabling the Execution of Extended BPEL Behavior” (described in Sect. 7.3.2) also allows for changing the behavior of BPEL and thus offers an alternative way to extend BPEL engines.

In the case that the system does not have adequate support, one must first enable it. This can only be done if the source code is accessible and can be modified, which is the case with the Eclipse BPEL designer and the Apache ODE engine, for instance. Consider a developer having an extension that introduces data references in BPEL during runtime execution (Wieland et al., 2009). The Eclipse BPEL Designer nor the Apache ODE engine supports this out of the box. Thus, the support has to be added to them by a programmer.

In all cases, commercial products are always a solution. Thus, the first decision to make is a make-or-buy decision (Jäger et al, 2008).

Commercial Solution: With most commercial workflow systems it is not possible to implement BPEL extensions, because their source code is not available and they do not provide an extension interface. Thus, only the usage of extensions provided by the vendor is possible. Nevertheless, a custom development of an extension by the vendor may be triggered.

Open Source Solution: The alternative is to implement a BPEL extension using an open source workflow engine. Compared to the hosted solution, this approach has the advantage that the developer has the full control over the development of the extension. Extensions are not restricted to defined extension points. If the system is running on a private server, execution of the extension can be observed and the data that is used in the workflow is secure (as long as critical data is not sent to external services). There are open source workflow engines available that can be used as development basis. As described in Sections 2 and 5, a modeling tool is also an essential part of the system and therefore has to be extended accordingly.

One of the goals of the standardization of BPEL has been the removal of all dependencies between process definition files, their process modeling tool, and the engines running those

[2] http://www.eclipse.org/bpel/

workflows. The modeling tool and the engine can be regarded as loosely coupled as they are replaceable by other systems that are implementing the BPEL 2.0 standard. This interchangeability breaks when a new extension activity is introduced. An extension activity typically enhances the set of BPEL activities and adds dependencies between the process engine and process modeling tool, as both have to understand how to handle these extension activities. Both systems (engine and modeling tool) have to care about the syntax of the extensions and the developer has to ensure that both systems rely on the same version of the extension activity. The engine needs to know what to do when it reaches the extension activity within a workflow model (semantics) and the modeling tool needs to know how to visualize, serialize and deserialize the activity to and from XML. Thus, both systems are not loosely coupled anymore. When creating a new extension activity, the workflow engine has to be extended via its extension API (if available). In addition, the modeling tool with its (mostly different) extension API has to be extended independently which leads to two extension implementations: one for the engine and one for the modeling tool. The developers have to take care that both versions do not differ in syntax and semantics.

For avoiding double implementation we developed a system design that allows using the same data model for the Eclipse BPEL Designer and Apache ODE (Fonden, 2009). This approach allows for implementing an extension by using a single shared Java class. The modeling tool and the engine use the corresponding parts of this class relevant for them: the modeling tool uses the layouting and XML serialization parts; the engine uses the execution code and the serialization code. This has the advantage that less inconsistencies, e.g., in the serialization or naming of the developed extensions, occur.

Hosted Solution: There is a current trend towards hosted “BPM as a service” systems, which are “Software as a Service” solutions targeting Business Process Management. As such, they provide a hosted system (accessible simply with a Web browser) for the end-to-end BPM lifecycle including design, execution, and monitoring. Additionally, such systems can enable collaboration between developers and designers. With nothing to install, this lowers the barrier to entry but does require a continuous connection to the Internet while working. In such systems, additional concerns arise for providing extensions. Referring back to the previous concerns, a developer has no access to modify the source and thus one must rely on supported extensibility. Thus, we focus on a concrete BPM as a service system presented by Curbera et al. (2007). It consists of a visual modeling tool backed by the Bite workflow runtime (Khalaf et al., 2009) and an extension catalog (Silva-Lepe et al., 2008). This system supports extensions and also enables collaboration around extension activities: Developers and designers can use the catalog to download, use, comment on, and rate extensions. The extension considerations highlighted in this section are the same for Bite and BPEL, because Bite’s control flow semantics are a subset of BPEL’s.

First, consider how the Bite runtime identifies and executes an extension activity: An extension is recognized upon encountering an unknown XML element in the process. The engine looks up a corresponding extension implementation module in an extension registry and associates it with the parsed activity. An extension implementation module may be written either in Java or in any of a set of supported scripting languages. When the extension activity is reached and activated in a process instance, the implementation module is called and handed the XML definition of the activity and required process instance data. The extension activity may only write data to the activity’s output variable. It does not have the ability to read or modify process navigational state. Once the implementation module completes, its output is stored in the output variable of the extension activity and the activity completes.

The extension enablement considerations for a hosted system include ensuring that the implementation artifact can reach the runtime, be registered in a catalog for use and looked up by the runtime and by other designers, and be able to be rendered by the design tool. In the system, a developer wanting to create an extension must provide basic activity metadata along with code implementing the extension. The meta-data is used by (a) the modeling tool, in order to provide the user with a meaningful display of the desired inputs for the extension and (b) the catalog, in order to provide a description and tags for users browsing the catalog. The implementation module is placed in a shared repository.

Developers upload the extensions either via a plug-in to their development environment or via a simple Web form. The extensions become immediately available to logged-in users. Once a user selects to use an extension activity in a workflow, its implementation module is pulled from the repository, the extension is registered with the engine and the module is bundled with the workflow application.

One key concern around extensions in a BPM as a service system is that it requires strong policing of the quality and integrity of the extension implementation code due to the fact that the environment is shared among many users and that the hosting entity may be liable for malicious extension code and potentially missed Service Level Agreements. This concern may be addressed by applying trust and reputation systems such as rating and ranking, third-party certification, and the ability to upload only by those with explicitly granted privileges.

7. Extensions in Practice

This section lists 62 commercially available extensions and scientifically published extensions. We apply the classification provided in Sect. 3. An extension may cover only the design time, the design time and the runtime, or only the runtime environment. The following is structured accordingly and additionally subdivided into vendor and research extensions.

7.1. Design Time only Extensions

This section presents approaches that make use of the transformation approach (A2) or that invoke dedicated services in order to integrate additional features (A3). It is also possible to combine both ways, as shown by Oracle's extensions presented in the following section.

7.1.1. Design Time only Extensions by Vendors

Oracle's Human Task (Oracle, 2007) activity is used to integrate human behavior into business processes (C1.8, C2.7). There are several configuration options, for example to route a task to a second approver or to execute a number of human tasks in parallel. Tasks can be assigned to humans by specifying concrete users or user groups. Depending on the chosen configuration human task activities are realized by `scope`, `assign`, `invoke`, `receive`, and `switch` activities. Oracle's Process Manager provides a dedicated human task Web service that is called by the `invoke` activity (C3.1, A3). A GUI enables the assigned user to handle the task (C4.1). The outcome of the task is sent back to the process. The extension mechanism used is an extension element that annotates the activities realizing the human task (s9).

Oracle's notification service (Oracle, 2007) is a collective term for five different notification mechanisms, namely email, fax, pager, SMS, and voice messages (C1.8, C2.7). Each is reflected by a single activity on a component palette in the process modeling tool (C4.1). Configuration of the activities is type-dependent. For example, the email activity provides parameters for target email addresses, a subject, and email body. The code underlying a notification activity is BPEL compliant: the activity is transformed into a `scope` with input, output, and fault variables, an `assign` to copy the user's parameter values to the input variable, an `invoke` activity to call a dedicated notification service, and a fault handler to deal with possibly occurring failures. An extension element annotates the `scope` to mark it as notification activity (s9). The appropriate notification service is provided by Oracle's Process Manager (C3.1, A3) that routes notifications to particular servers (email server, SMS server, etc.).

7.1.2. Design Time only Extensions by Research

BPEL4Chor is extending BPEL with a unique ID which is used for identifying message activities and `onMessage` branches. Decker et al. (2009) present *BPEL4Chor* as an extension of BPEL for modeling choreographies. A choreography describes the message exchange between multiple participants (Peltz, 2003; C1.1, C4.1). BPEL4Chor uses BPEL to describe the behavior of each participant in a choreography (C2.1, C3.2). The BPEL4Chor topology lists the participants and the connection between them in the form of message links. A unique ID is used to identify the activities and `onMessage` branches, which are referenced in a message link. The ID is stored in the attribute `wsu:id` (s8). The `name` attribute is not used since it is not possible to put a `name` attribute on an `onMessage` construct. Each participant behavior description is transformed to an abstract BPEL process following the abstract process profile for observable behavior. This model does not contain any extensions any more. The model is then manually refined to an executable BPEL process without addition of any BPEL4Chor related extensions (C4.2). This makes BPEL4Chor a design time only extension.

The *ID attribute* is a general extension where a unique identifier may be put to each element in the BPEL processes (s8). The identifier is mainly used in modeling, such as for referencing particular constructs (C1.4, C2.5, C3.2, C4.1). The modeling extension does not need to be understood by the engine (`mustUnderstand="no"`) since there is no runtime behavior of the identifiers.

BPEL process templates (Karastoyanova, 2006) are abstract, reusable units of BPEL code stored in a separate XML file (`*.template`). Usually, a template solves a general, recurring problem that can be used to avoid process modeling from scratch and reinventing the wheel (C1.2, C1.6). Templates are abstracted with the help of parameters that hide certain details (e.g., variables, partner links, port types). At buildtime, parameters can be mapped on concrete values provided by the process modeler. Templates can be referenced from within BPEL processes by a `tRef` element in BPEL namespace (C2.5, $\overline{s}$2). Using such references in templates allows recursive template definition (C3.2, C4.1). Since processes pointing to templates are not executable, transformation steps need to be performed in order to make them executable. Template parameters are thereby substituted by concrete values, template references by the actual template code (A2, C4.4).

"SWRL for BPEL" (Wu et al., 2008) defines how constraints between BPEL activities can be encoded in the BPEL process using the Semantic Web Rule Language (SWRL). This enables another way of modeling process models (C1.4, C2.1, C3.2, C4.1). The extended BPEL process is transformed to a standard-conform process following the given constraints (A2). The extension declares new extensions elements (s9).

BPEL fragments (Ma et al., 2007) are introduced as modeling construct to enable reuse of process parts across different processes (C1.6, C2.1, C3.2, C4.1). The approach does not use BPEL's extension mechanisms, but declares a new namespace and uses `fragment` instead of `process` as root element ($\overline{s}$5).

BPEL-D (Khalaf and Leymann, 2006) replaces variables by explicit data links in BPEL 1.1 (C2.8, C3.2, C4.1). In general, there are two ways to propagate data between activities in business processes: the blackboard approach and explicit data flow (Alonso et al. 2003, p. 266). In the case of the blackboard approach, variables are used to share data. BPEL implements the blackboard approach, whereas BPEL-D realizes explicit data flow. Thus, BPEL-D contradicts with the BPEL semantics ($\overline{s}$4). The motivation of BPEL-D is enabling business process outsourcing (C1.1): A BPEL-D process is used as input for an algorithm splitting the process into several standards-conform BPEL processes, which maintain the operational semantics of the intended BPEL-D process (Khalaf, 2008). Thus, BPEL-D is only used at design time. It is possible to transform one BPEL-D process into one standard BPEL process reassembling BPEL-D semantics by standard BPEL constructs.

BPEL data transitions (BPEL-DT) extend the BPEL language with data transitions for handling large amounts of data (Habich et al., 2007; C2.4). This is, for instance, required in ETL (extract, transform, load data) flows that are based on Web service orchestrations which are realized with BPEL. Such data intensive service applications can make only limited use of the "by value" semantics in BPEL, as otherwise massive data sets have to be transferred forth and back to the process engine. Other ways of specifying data flow are therefore necessary. In standard BPEL, data flow is implicitly contained by the access of activities to variables and their values, respectively. BPEL-DT seeks to make data flow explicit by extending the BPEL metamodel with data transitions (i.e.,data links; C1.3, C1.5, C1.8, C4.1). These links are transformed into an XML mapping specification (A2; MSL, IBM, 2007), which needs to be manually refined (C4.2). The engine then calls additional services to realize the given mapping specification (A3, C3.1, C4.5). This extension is not implemented in a standard-conform manner and contradicts the BPEL semantics, since a new kind of links is added ($\overline{s}$4). In BPEL-D, data-flow is still internal to the process, whereas BPEL-DT externalizes the data flow.

References in BPEL (Wieland et al., 2009) also address handling large amounts of data by extending BPEL's data handling mechanism with pointers on data (C1.8, C2.2, C2.4). A BPEL `referenceVariable` element in BPEL namespace ($\overline{s}$2) is introduced that specifies variables containing a reference to externally stored data (C3.2). The attribute `referenceType` indicates whether a reference is resolved at `scope` activation, before each usage, periodically, or on behalf of an external partner (C2.8). Actual reference resolution is made by an external Reference Resolution Service (RRS) (C3.1, A3). Since "References in BPEL" is proposed as build time extension, a pre-deployment step needs to transform extended BPEL files into standard BPEL by replacing reference variables with BPEL variables, inserting partner links and interaction activities (depending on the reference type) (C4.1, C4.4, A2).

"*Activity failure and recovery*" is a BPEL extension proposed by Liu et al. (2007) which is intended to increase the reliability of processes and to relieve process modelers from the complexity of defining BPEL fault handlers. They therefore introduce four fault tolerance patterns (ignore fault, skip scope, retry scope, and alternative scope) that can be exploited during modeling of processes to express reactions on faults (C4.1). The specified patterns are not included in the designed process but are mapped on `scope`s by name. Each pattern consists of rules to transform a given process definition into a process that implements the particular fault tolerance mechanism (e.g., retry a scope a specified number of times) (C1.7, C2.1, C3.2).

"Activity failure and recovery" is also proposed by Modafferi and Conforti. (2006). Here, an annotated BPEL process is used as starting point. The annotations include setting variables by external messages (C1.2, C2.2, C3.2), specifying timeouts for service invocations (C1.7, C2.7, C3.2) and enabling redoing of an activity (C1.2, C1.3, C3.2). The annotated process is then transformed to a standard BPEL process (C4.1, A2). The extensions are put in the BPEL namespace ($\overline{s}$ 2).

xBPEL (Chakraborty et al., 2004) is a BPEL extension for modeling mobile participants in workflows (C1.2, C1.3). Chakraborty et al. introduce the PerCollab system which executes xBPEL and allows mobile integration of people into BPEL workflows without constraining the users to their desktop PC. xBPEL allows modeling communication between people and between a process and people (C2.1, C3.3, C4.1, C4.5). The extensions are put into the BPEL namespace ($\overline{s}$ 2). An xBPEL process is transformed to standard BPEL process (A2) and services of the PerCollab environment (A3).

7.2. Design Time and Runtime Extensions

This section lists extensions, where the BPEL modeling tool and the BPEL runtime are extended.

7.2.1. Design Time and Runtime Extensions by Vendors

WS-BPEL Extension for People (*BPEL4People*) enables integration of human-based activities in BPEL (Agrawal et al., 2007a). This includes the possibility to define people's activities, people groups, tasks and notifications (C1.3, C2.1, C3.1, C3.2, C3.3, C4.1, C4.5). BPEL4People is building on WS-HumanTask (cf. Agrawal, 2007b). WS-HumanTask is used in BPEL4People for the actual implementation of a people activity. BPEL4People defines the `peopleActivity` as a basic activity type which uses human tasks as an implementation (C2.7, s7). The `peopleActivity` allows specifying tasks local to a process or use tasks defined outside of the process definition. To use BPEL4People the modelling tool and the process engine must be extended (A1, A3).

BPEL for Java (*BPELJ*) combines the programming languages BPEL and Java (Blow et al., 2004). The intention is to provide a way for integrating pieces of Java code into a BPEL process definition. The main effect of this extension is a higher convenience when programming a BPEL process (C1.3, C1.5, C1.8). BPELJ allows using Java code to be included in BPEL process definitions. The according activity in BPELJ is named `snippet`. In a snippet, BPEL variables can be manipulated and those snippets can be used for instance in loop conditions, branching conditions (C2.1) and for variable initialization as well as variable manipulation (C2.2, C2.3, C2.8). To use BPELJ extended modeling tools and process engines must be implemented (A1). Since BPELJ allows the modification of variables in a transition condition, it is not conform to the BPEL execution semantics ($\overline{s}$ 4).

BPEL-SPE (Kloppmann et al., 2005) is a BPEL 2.0 extension for sub-processes that aims at increasing legibility and reusability of processes (C1.4, C1.6, C1.8). Sub-processes are BPEL processes implementing a single request-response operation and are called using a `call` activity in BPEL namespace from within the parent process (C2.5, C2.7, C4.1, $\overline{s}$ 2). The life cycle of sub-processes is tied to the respective parent process (C1.3, C3.2). For instance, a fault in a sub-process needs to be propagated to the parent process. This is enforced by coordination messages employed BPEL engines need to understand (A1, C4.4, C4.5, C4.6). Sub-processes can be defined as standalone process (C1.1) and inline within a parent process (C3.2). An inline sub-process can access visible data (i.e.,data of the scope it is defined in) of its parent process and thus omit implementation details.

The *Execution as Subprocess* extension (IBM, 2009) is a variant of BPEL-SPE. The goal is to enable an execution as a subprocess in a declarative way instead of a `call` activity (C1.3, C2.1, C2.7, C3.1, C3.2, C4.1, C4.5). The partner link declaration is extended by the attribute `processTemplate` (s8). Here, the name of a BPEL process may be specified. If the execution engine finds that process at the runtime, the process is directly called by the BPEL engine and the life cycle of the process is tied to the caller (A1). That means, for example, that a fault on process level of the called process is communicated to the calling process.

The *Collaborative Scopes* approach (IBM, 2009) adds support for case handling (van der Aalst et al., 2004) to BPEL processes (C1.3, C2.1, C3.1, C3.2, C4.5). A new `collaborativeScope` activity is introduced (C4.1, s7). Each activity in a collaborative scope may have an exit condition. It is possible to evaluate the exit condition on start or on completion of an activity, or both. In case the condition is evaluated at the start, the activity is skipped if the exit condition is met. In case the exit condition is evaluated at the completion of an activity and the exit condition evaluates to false, the activity is started again. The extension is included in the modeling tool and realized in the engine (A1).

The *Generalized Flow* (IBM, 2009) enables control links to connect activities arbitrarily (C1.3, C1.8, C2.1, C3.2, C4.1, C4.5, $\overline{s}$4). Standard BPEL allows links to form an acyclic graph only. In addition to arbitrary connections, fault links between two activities are introduced. If the source activity faults, the target activity is executed. The generalized flow has to consist of one start activity only and only one control link may be followed at each execution step. The approach requires an engine extension (A1).

ii4BPEL (IBM, 2006) integrates SQL statements into BPEL, connects processes directly to relational databases, and supports advanced ways of data exchange (C1.3, C1.5, C2.2). IBM implements ii4BPEL in the WebSphere Integration Developer as a Plugin. Based on BPEL 2.0 IBM extended the BPEL language and the tooling, e.g., the process engine, the deployment mechanism, the modeling tool (A1, A2, C3.1, C3.2, C4.1, C4.4, C4.5). Furthermore, a special data middleware is required (A3). ii4BPEL defines four new activities for data management (s7): `SQLSnippet` runs an SQL statement against database tables. `retrieveSet` load referenced data sets into BPEL-variables. `atomicSQLSequence` join SQL snippets and retrieve sets in one activity. `informationServer` interacts with the IBM InfoSphere Information Server.

Non-compensatable scopes (IBM, 2009) introduces the attribute `compensatable` to a scope. In case the attribute is set to `yes`, a compensation of the scope leads to a fault (A1, $\overline{s}$4). The feature is used to improve performance of process execution: In case a scope is marked as non-compensateble, no snapshots of variables after the completion of the scope are needed (A1, C1.3, C2.5, C3.1, C4.1, C4.5).

Dedicated Administrator (IBM, 2009) enables the assignment of an administrator to a scope at the beginning of its life cycle. The administrator may do corrective changes to variables and has full control over the life cycle of the scope to ensure proper process execution (A1, C1.7, C2.5, C3.1, C4.1, C4.5, s8).

A *microflow* (IBM, 2009) is a new execution mode for business processes indicated by `wpc:executionMode="microflow"`. A microflow is a micro script which is executed in one transaction to speed up processing (Leymann & Roller, 2000; C1.5, C2.5). Due to the single transaction, the starting receive is the only receive allowed. Asynchronous invokes are always allowed, whereas synchronous invokes only in the case of synchronous bindings (C3.1, C4.1, C4.5).

Transaction boundaries (IBM, 2009) enable configuration of the internal behavior of the BPEL engine with respect to its internal atomic transactions (A1, $\overline{s}$4). The navigator of a BPEL engine usually starts a new transaction at the beginning of an activity and commits it at the end of the activity. This execution causes an overhead at the transaction manager. By configuring the transactions to span multiple activities, this overhead and hence the process execution time can be reduced (C1.5, C2.5, C3.1, C4.1, C4.5).

The Apache ODE group (2009) proposes eight extensions to facilitate execution of BPEL processes. The specification of these extensions does not require declaration of the extensions. Besides adding new activities and attributes, the Apache ODE engine[3] offers support for XPath 2.0 as query language and adds *new XPath functions* reducing the coding effort. For instance, the function `insert-before` inserts a node as a sibling before a given node (C1.3, C2.5, C3.2, C4.1, C4.5).

Implicit correlations remove the need to add correlation sets in the case the BPEL process starts the interaction with a service (Apache ODE group, 2009). By using implicit correlation, a unique session identifier is generated and put into the message (C1.3, C2.7, C3.1, C4.1, C4.5, $\overline{s}$4). The response of the service contains the same session identifier. The message router of the engine uses

[3] http://ode.apache.org/

this identifier to route the message to the correct process instance. A concrete implementation is available for the SOAP/HTTP binding (A1).

Activity failure and recovery enables configuration of failure handling in the case of an invoke activity (Apache ODE group, 2009). An example for a failure is an HTTP timeout. Default failure handling shows faults in the process instance management of Apache ODE and requires manual intervention. This behavior can be changed by a `failureHandling` element ($\overline{s}$4). It can be configured as follows: `retryFor` specifies the number of retries; `retryDelay` denotes the time between each retry; `faultOnFailure` causes the invoke activity to throw an `activityFailure` fault as BPEL standard fault in the case of a failure (A1, C1.7, C2.7, C3.1, C4.1, C4.5).

Headers handling enables the access to header fields in SOAP messages (Apache ODE group, 2009; A1). For that purpose, the attribute `header` is introduced into the BPEL namespace at the `from` and `to` elements of a `copy` statement in an assign activity ($\overline{s}$3). In case the attribute is present, the context node of the XPath statement is set to the specified header element (C1.8, C2.1, C3.2, C4.1, C4.5). There is no explicit possibility to check for presence or absence of header fields.

The *iterable forEach* adds the element `sequenceValue` to the BPEL namespace below a BPEL `forEach` (Apache ODE group, 2009; A1, $\overline{s}$3). If the element is present, the `forEach` iterates on all elements contained in the given `xsd:sequence` element instead of using start and final counter value (C1.3, C2.1, C3.2, C4.1, C4.5, $\overline{s}$4).

The *auto complete copy destination* enables the attribute `insertMissingToData` in a `to` statement `copy` statement in an `assign` activity (Apache ODE group, 2009; $\overline{s}$3). If set to `yes`, the path to the element given in `to` element of a `copy` statement is automatically generated (C1.3, C1.4, C2.1, C3.2, C4.1, C4.5). For example, if `New York` is assigned to `$customer/address/city`, but the variable `$customer` is empty, the parent elements `address` and `city` are automatically generated.

To enable *ignoring unavailable data* the two attributes `ignoreMissingFromData` and `ignoreUninitializedFromVariable` are introduced to the `copy` statement of the `assign` activity (Apache ODE group, 2009; $\overline{s}$3). In the case of `ignoreMissingFromData` and a from-spec returning no XML information items, the `selectionFailure` fault is suppressed and no assignment done. In case of `ignoreUninitializedFromVariable` and the usage of an uninitialized variable in the from-spec, the `uninitializedVariable` fault is suppressed and no assignment is done (A1, C1.3, C1.4, C2.1, C3.2, C4.1, C4.5).

Process contexts are key value pairs allowing metadata in sent and received messages to be stored and accessed in processes (C1.3, C1.4, C2.1, C2.2, C3.2, C4.1). The contexts can be used in assign activities and in invoke activities (A1, $\overline{s}$4). Developers have to provide Java code to copy SOAP header information from and to context objects in Apache ODE. The Java code compiled and stored in the engine. The functionality is activated using `properties`-files and `deploy.xml`.

Resource-oriented BPEL is an approach to add support for providing and using REST services in BPEL. The Apache ODE group and Overdick (2003) propose to add special REST attributes to the `invoke` activity, the `receive` activity and the event handler (C1.3, C1.4, C1.8, C2.1). That way, RESTful services are directly supported by BPEL instead of using a special HTTP binding in WSDL.

BPEL for REST is an approach shown in Pautasso (2008). Four activities (`get`, `put`, `post`, and `delete`) are used to invoke REST services ($\overline{s}$2, $\overline{s}$4). RESTful resources can be offered via `onGet`, `onPut`, `onPost`, and `onDelete` handler (A1, C1.3, C2.1, C2.6, C3.1, C4.1, C4.5).

Continue on error (IBM, 2009) offers a similar behavior as activity failure and recovery. Each invoke activity gets the attribute `continueOnError` (s8). A human task for an administrator is generated in the case the invoke activity encounters a communication failure and the value of the attribute is `yes`. The assigned administrator is then privileged to do corrective actions. In the case of a `no`, the failure is converted into a fault and thrown into the BPEL process (C1.7, C2.7, C3.1, C4.1, C4.5).

7.2.2. Design Time and Runtime Extensions by Research

Retry scopes (Eberle et al., 2009) extend BPEL with scope retrying behavior (C1.3, C1.7, C2.1, C3.2, C4.1, C4.5). The idea is similar to the idea presented by Liu et al. (2007). In Eberle et al. (2009), the issue of retrying is solved with an explicit `restart` activity and without an à priori rewriting step (A1, $\overline{s}$4). The `restart` activity may only be used in a fault handler and restarts the respective scope. By using an explicit activity, explicit repair behavior may be executed before restarting the scope.

BPEL/SQL (Vrhovnik et al, 2007) is a generic term for approaches to integrate SQL statements into BPEL with the aim to connect workflow engines directly to relational databases. Vrhovnik et al. (2008) have presented an overview of BPEL/SQL implementations, which all share the properties of ii4BPEL described in Sect. 4.2.1: A1, A2, A3, C1.3, C1.5, C2.2, C3.1, C3.2, C4.1, C4.4, C4.5, s7.

Parameterized processes (Karastoyanova, 2006) is an extension that decouples BPEL's interaction activities from concrete port types and operations to improve reusability of (parts of) workflows (C1.6) and flexibility of selecting arbitrary services at runtime (C1.2). The new element `evaluate` is inserted under BPEL namespace into message sending activities to override the specified port type/operation pairs (C3.2, C4.1, $\overline{s}$2). The "evaluate" concept enables several strategies to provide an activity with a concrete port type/operation (static, prompt the user, query, and from variable) (C4.5, C4.6). The approach allows determining the interface of the service to invoke at runtime, taking different interfaces for different process instances, or handling faulty Web service invocations by default port type/operation pairs (C1.7, C2.7). In conjunction with the "evaluate" extension the `find_bind` element is introduced (in BPEL namespace) which can be used in message sending activities (C3.2, C4.1 $\overline{s}$2). It enables a deployment-independent specification of service selection policies even at runtime (C4.4), the runtime modification of such policies even for single process instances (C132) as well as a process instance repair if the service selection fails (C2.6). The parameterized processes approach extends both design time and runtime environments (A1).

Cross-process fault handling and transaction handling (Kopp et al., 2009) enables grouping arbitrary activities of different participating processes together to form a logical transaction unit called choreography sphere (C1.3, C1.7, C2.1, C4.1). The grouping and additional handlers are specified outside the BPEL processes in the choreography. To execute the choreography sphere, an additional coordination infrastructure is needed (C3.1). Thus, the runtime semantics of BPEL is changed (C3.2, C4.5, $\overline{s}$4).

The *E4X extension for BPEL* (van Lessen et al., 2009) enables the usage of ECMAScript for XML (E4X; International Organization for Standardization, 2006) instead of XSLT and XPath in the case of variable manipulation. E4X extends JavaScript with support for XML-based data manipulation (C1.4, C2.3, C3.2, C4.1, C4.5). The extension defines an `extensionAssignOperation` and an `extensionActivity`, where JavaScript code may be used (s7, s10).

Context4BPEL (Wieland et al., 2007) allows the definition of context-aware workflows (C1.3). Such workflows may be used to create context-aware applications or to apply workflow technology in manufacturing production processes, for example (C4.6). Context4BPEL provides several extensions in a `c4b` namespace to implement three concepts for explicitly making use of context information from within workflows. First, the workflow can handle context events by particular activities that register (`c4b:registerSpatialEvent`), deregister (`c4b:deregisterSpatialEvent`) and update (`c4b:updateSpatialEvent`) events (C2.1, C2.7, C3.2, s9). Context events can be received by any incoming message activity with certain message types. Second, context data can be queried by a `c4b:queryContext` activity that stores the result of the request in a variable with well-defined type. Third, transition conditions can be evaluated based on workflow internal or external context data (C2.2, C2.3). New XPath functions are specified that facilitate dealing with context information, e.g., the `c4b:within(area, location)` function. Context4BPEL extends both design and runtime environment (C4.1, C4.5, A1).

BPEL4Grid (Dörnemann et al., 2007) combines workflow and grid technology. The extensions help to invoke stateful Grid services (C1.3, C1.4, C2.2, C2.7, C3.1, C3.2, C4.1, C4.5). BPEL4Grid defines three new activities: `GridInvoke`, `GridCreateResourceInvoke`, `GridDestroyResourceInvoke`. Since BPEL4Grid introduces an additional way to communicate with services, it is not standards compliant ($\overline{s}$2). BPEL4Grid includes an extended modeling tool and an extended process engine (A1). A similar approach is presented by Zhang et al. (2008) where a `GrsService` activity is used to call a stateful Grid service.

BPEL^{light} (Nitzsche et al., 2007a) is an extension of BPEL 2.0 that decouples process logic from WSDL 1.1 interface definitions to improve reusability of process models and to enable workflow modeling without WSDL knowledge (C1.2, C1.6, C1.8, C4.1). BPEL^{light} introduces a novel interaction model with the help of BPEL's extension activity mechanism (C1.3, C3.2, C4.5): The WSDL-less `bl:interactionActivity` emulates the behavior of `receive`, `reply`, and `invoke` activities (C2.7). WSDL-less `bl:pick` and `bl:eventHandlers` replace their BPEL counterparts. BPEL's partner link concept is split to BPEL^{light} `bl:partners`, containers for partner endpoint references (EPRs), and `bl:conversations`, message exchanges that can involve several messages and partners ($\overline{s}$ 4 – contradicts BPEL's communication paradigm). Interaction activities can be arbitrarily bound to synchronous or asynchronous services (C1.2, C2.6, C4.4). BPEL^{light} results in an extension of design time and runtime environment (A1).

BPEL for Semantic Web Services (BPEL4SWS) by Nitzsche et al. (2007b) proposes WSDL-less BPEL by removing these artifacts and thereby increasing the flexibility of business processes. In contrast to BPEL^{light}, BPEL4SWS uses semantic web technology, whereas BPEL^{light} uses straight-forward communication paradigms. BPEL4SWS uses a set of composable standards and specifications and is independent of any Semantic Web Service framework. It can be used to compose Semantic Web Services, traditional Web Services and a mix of them (A1, $\overline{s}$ 4, C1.2, C1.3, C1.6, C1.8, C2.6, C2.7, C3.2, C4.1, C4.4, C4.5).

OWL for BPEL integrates semantics in the form of OWL to BPEL (Le et al., 2009). Messaging activities are replaced by generic `ontcaf:service` element, which directly specifies its input and output data formats ($\overline{s}$ 4). The integrated OWL information is used to find a matching service for each specified service (A1, C1.2, C2.1, C3.1, C4.1, C4.5).

WS-BPEL Extensions for Versioning (Juric et al., 2009) addresses the problem of versioning BPEL processes and partner links (C1.4, C2.5, C3.1). The extension introduces new activities such as `versionHandlers` and adds attributes to existing activities such as `invoke`, `receive`, `import`, or `onMessage` in the BPEL namespace ($\overline{s}$ 2). It also extends the partner links concept at different levels of versioning. To use BPEL for Versioning the modeling tool, the process engine and the deployment mechanism must be upgraded (A1, C4.4).

"*BPEL for pervasive computing*" (Hackmann et al., 2007) introduces a multicast and publish/subscribe mechanism in BPEL 1.1 (C1.3). The aim is to make BPEL usable in pervasive and mobile computing scenarios where peers can enter or leave the network at any time and hence the number of message recipients is unknown at design time (C1.8). A new `ext:partnerGroup` construct works as list of endpoint references (EPRs). Management of this list is realized by `ext:add` and `ext:remove` activities to insert or delete EPRs, respectively. The `ext:reply` activity can exploit a partner group to send messages to all contained partners, eventually realizing a multicast (C2.7, C3.2, C4.1, C4.5). Since several partners communicate with the process over one and the same partner link, there is a need to explicitly unbind a partner link (`ext:unbind` activity) and close its connection (`ext:close` activity) ($\overline{s}$ 4 – contradicts BPEL's communication paradigm). The approach requires a design time and runtime extension (A1).

T-BPEL (Tai et al, 2004) stands for "Transactional BPEL" and allows for attaching transaction requirements to a BPEL process and transaction capabilities to Web services. This enables a BPEL process to initiate distributed atomic transactions as well as compensation based transactions (C1.3,C2.7,C3.1,C4.1,C4.5). The extension is fully BPEL 1.1 compliant as it uses a separate namespace for its attributes (s8) and does not change the behavior of the BPEL engine.

7.3. Runtime only Extensions

In the case of a runtime only extension, the process model itself stays unchanged but other artifacts are touched, e.g., the deployment descriptor is modified. Runtime only approaches are not an extension in terms of Definition 1. We show them to emphasize the difference between a language extension and other forms of modifications and use the term "extension" for consistency with the terminology of the workflow community.

Runtime only extensions involve particular new components, but they have no impact on the modeling tool. It is possible, however, that such an extension offers other modeling tools for their particular purpose, different from BPEL modeling tools.

7.3.1. Runtime only Extensions by Vendors

"Business rules integration" is presented in Oracle (2006). Here, a business rule engine can be used within a BPEL process by using the `invoke` activity which calls a dedicated service (A3) for processing business rules (C1.2, C1.4, C1.6, C2.1, C3.2, C4.1, C4.5). This service interacts with a rules engine, which again is integrated with a rule authoring tool and a rules repository. For evaluation of a rule all required parameters are passed in the actual service call. The result of the business rule service invocation can then be used in further processing, e.g., as `transitionCondition` on a control link.

7.3.2. Runtime only Extensions by Research

BPEL'n'Aspects (Karastoyanova and Leymann, 2009) is an approach of applying the aspect-oriented programming (AOP) paradigm (Kiczales, 1997) to BPEL processes to facilitate adaptations of running service compositions (C1.2). It enables to insert (or weave) aspects into processes without touching these processes themselves. Aspects are described by WS-Policy (Vedamuthu et. al, 2007a). They contain a pointcut (i.e.,the place in the process to weave the aspect in) and an advice (i.e.,the functionality to weave in). Possible pointcuts are described by joinpoints that can currently be activities and transition conditions. In BPEL'n'Aspects, an advice is always a Web service invocation (C2.7). There are three advice types that denote whether the invocation ought to be carried out before, instead, or after a BPEL construct. Aspects are weaved into processes with the help of the WS-Policy Attachment mechanism (Vedamuthu et al., 2007b). BPEL'n'Aspects enables to insert aspects into single process instances, process instance groups, or all process instances of a process model (C1.4, C1.6, C3.2). The actual weaving can be done at runtime by the BPEL engine itself or by an external component (i.e.,the weaver) on basis of appropriate events created during workflow execution (C4.5). Since the engine itself is not aware of the executed aspects, the auditing needs to be extended in order to provide compensation capabilities.

AO4BPEL (Charfi and Mezini, 2004) is an approach similar to BPEL'n'Aspects, but enables BPEL snippets to be weaved into (running) processes (C1.2, C1.4, C1.6). Aspects are expressed as BPEL extension in BPEL namespace with an `aspect` element ($\overline{s}$ 2, $\overline{s}$ 4). Pointcuts are XPath expressions contained in a `pointcut` element. Each BPEL activity is thereby a possible joinpoint (C3.2). Advices are BPEL snippets nested in an `advice` element (C2.1, C2.7). An AO4BPEL implementation foresees an extended aspect-aware BPEL process engine and an aspect manager which execute activated aspects (C4.5).

A variant of *activity failure and recovery* is presented in Juhnke et al. (2009) and Wen et al. (2006). They propose to change the way of service invocation to support handling of unavailable services by retrying invocation or replacing the called service (C1.7, C2.7, C3.1, C4.5). Both assume that the service is idempotent and that each operation implements an in/out operation. Both add a new deployment artifact which specifies a policy for handling a network fault.

A second variant of *activity and failure recovery* is presented in Kareliotis et al. (2007). There, a transformation of a BPEL process is proposed. Each `invoke` in the input BPEL process is surrounded by a fault handler. In the case of a transportation fault, a service registry is invoked. The registry returns compatible services (C1.7, C2.7, C3.1, C4.5). Each service of the list is tried to be invoked subsequently until an invocation succeeds. The original BPEL process does not need to be modified. The generated BPEL process requires a service registry. Thus, we treat the extension as a runtime only extension, although the behavior of the transformed BPEL process does not rely on an extended BPEL engine.

SH-BPEL is a variant of "activity failure and recovery" (Modafferi, Mussi, Pernici, 2006) shows an enhancement of the invocation handler of a BPEL engine to support failure handling in the engine. Such failure handling includes replacing a service or to trigger human involvement. This extension is not an extension in our sense, since the runtime of BPEL is changed without any change of the BPEL process (C1.7, C2.7, C3.1, C4.5).

"*Extended WS-RM*" (Charfi et al., 2009) also deals with reliability. In their case, they extend WS-Reliable Messaging (WS-RM; OASIS, 2004) to support multi-party conversations specified in BPEL (C1.3, C2.7, C3.1, C4.5). WS-RM is a standard used to realize reliable messaging requirements on a SOAP level (Weerawarana, 2005). The extension is implemented in the invocation handler. The behavior of the invocation handler is configured by the deployment descriptor.

The *"Pluggable Framework for Enabling the Execution of Extended BPEL Behavior"* (Khalaf et al., 2008) offers a systematic mechanism to instrument BPEL engines so that behavior can be injected into a process (C1.2, C2.1, C2.2). The framework is based on a generic event model which can be mapped to lifecycle events of particular BPEL engine. These events are forwarded to a custom controller (C3.1), which can execute arbitrary behavior (e.g., require by an extension). The event may be a 'blocking event', in which case navigation is suspended on the respective path in the process until it receives an unblocking notification from the controller. Data in this notification may potentially affect how the navigation in the process proceeds. The additional behavior is effective during the execution of a process (C4.5).

"A Management Framework for WS-BPEL" (van Lessen et al., 2008) has the same aim as the pluggable framework (C1.2, C2.1, C2.2, C3.1, C4.5). In contrast to rely on events, it renders the activities of the BPEL process as resources and thus offers a uniform access scheme.

"*Business Rules Integration in BPEL*" (Rosenberg and Dustdar, 2005) makes use of interceptors to trigger business rule checks. Interceptors can be attached before or after message sending/receiving activities. This mapping of interceptors on BPEL activities is provided by the person who models the process. That way, business rule definitions are separated from process logic (C1.2, C1.4, C1.8). An extended enterprise service bus (ESB) interprets the mapping and executes the business rules (C4.5). Negative evaluated rules cause the respective activity to be skipped (C2.1, C3.2). A transformation engine for message mediation and a rule broker allow the integration of different rule engines.

7.4. Summary

We discussed a huge variety of extensions addressing different aspects of the BPEL environment (Figures 1 and 2). Tab. 4 presents an overview of the extensions discussed including a characterization in terms of the criteria introduced in Sect. 3. The table has six columns: The column extension lists the name of the extension; Extd (Language Extended) states whether the BPEL language has been extended with any new construct; Conform states whether the extension is conform to Definition 1 using the standard conformity shortcuts of Tab. 1; A (Approach) lists the approaches that were applied (cf. Sect. 5); Type lists D or R denoting the type of the extension: D stands for a design time extension, R stands for a runtime extension; Characteristics lists the characteristics of the extension (referring to Tab. 3).

Our classification is based on a literature study. We did not interview the extensions authors to find out the thoughts behind their extension design. We assume that all the authors fulfilled their goals as their extensions are available. When the authors followed our extension development guidelines presented in Sect. 6, they would have possible chosen another way. For instance, for enabling a retry of failed calls, the invocation handler could be modified.

Tab. 4: Extension Overview

Extension	Extd	Conform	A	Type	Characteristics
A Management Framework for WS-BPEL (van Lessen et al., 2008)	No	n/a	n/a	R	C1.2, C2.1, C2.2, C3.1, C4.5
Activity failure and recovery (Apache ODE group, 2009)	Yes	No ($\overline{s}$ 4)	A1	D, R	C1.7, C2.7, C3.1, C4.1, C4.5
"Activity failure and recovery" (Juhnke et al., 2009; Wen et al., 2006)	No	n/a	n/a	R	C1.7, C2.7, C3.1, C4.5
"Activity failure and recovery" (Kareliotis et al., 2007)	No	n/a	n/a	R	C1.7, C2.7, C3.1, C4.5
"Activity failure and recovery" (Liu et al., 2007)	No	n/a	n/a	D	C1.7, C2.1, C3.2, C4.1
"Activity failure and recovery" (Modafferi & Conforti, 2006)	Yes	No ($\overline{s}$ 2)	A2	D	C1.2, C1.3, C2.2, C3.2, C4.1
AO4BPEL (Charfi and Mezini, 2004)	Yes	No ($\overline{s}$ 2, $\overline{s}$ 4)	n/a	R	C1.2, C1.4, C1.6, C2.1, C2.7, C3.2, C4.5
Auto complete copy destination (Apache ODE group, 2009)	Yes	No ($\overline{s}$ 3, $\overline{s}$ 4)	A1	D, R	C1.3, C1.4, C2.1, C3.2, C4.1, C4.5
"BPEL for Pervasive Computing" (Hackmann et al., 2007)	Yes	No ($\overline{s}$ 4)		D, R	C1.3, C1.8, C2.7, C3.2, C4.1, C4.5
BPEL for REST (Pautasso, 2008)	Yes	No ($\overline{s}$ 2, $\overline{s}$ 4)	A1	D, R	C1.3, C2.1, C2.6, C3.1, C4.1, C4.5
BPEL fragments (Ma et al. 2009)	Yes	No ($\overline{s}$ 5)		D	C1.6, C2.1, C3.2, C4.1
BPEL process templates (Karastoyanova, 2006)	Yes	No ($\overline{s}$ 2)	n/a	D	C1.2, C1.6, C2.5, C3.2, C4.1, C4.4
BPEL/SQL (Vrhovnik et al, 2007)	Yes	Yes (s7)	A1, A2, A3	D, R	C1.3, C1.5, C2.2, C3.1, C3.2, C4.1, C4.4, C4.5
BPEL'n'Aspects (Karastoyanova and Leymann, 2009)	No	n/a	n/a	R	C1.2, C1.4, C1.6, C2.7, C3.2, C4.5

Extension	Extd	Conform	A	Type	Characteristics
BPEL4Chor (Decker et al., 2009)	Yes	Yes (s8)	A2	D	C1.1, C2.1, C3.2, C4.1, C4.2
BPEL4Grid (Dörnemann et al., 2007 and Zhang et al., 2008)	Yes	No ($\overline{S}$ 2)	A1	D, R	C1.3, C1.4, C2.2, C2.7, C3.1, C3.2, C4.1, C4.5
BPEL4People (Agrawal et al., 2007a)	Yes	Yes (s7)	A1, A3	D, R	C1.3, C2.1, C2.7, C3.1, C3.2, C3.3, C4.1, C4.5
BPEL4SWS (Nitzsche et al., 2007b)	Yes	No ($\overline{S}$ 4)	A1	D, R	C1.2, C1.3, C1.6, C1.8, C2.6, C2.7, C3.2, C4.1, C4.4, C4.5
BPEL-D (Khalaf and Leymann, 2006)	Yes	No ($\overline{S}$ 4)	n/a	D	C1.1, C2.8, C3.2, C4.1
BPEL data transitions (BPEL-DT; Habich et al., 2007)	Yes	No ($\overline{S}$ 4)	A2, A3	D, R	C1.3, C1.5, C1.8, C2.4, C3.1, C4.1, C4.2, C4.5
BPELJ (Blow et al., 2004)	Yes	No ($\overline{S}$ 4)	A1	D, R	C1.3, C1.5, C1.8, C2.1, C2.2, C2.3, C2.8, C3.2, C4.1, C4.5
BPEL^{light} (Nitzsche et al., 2007a)	Yes	No ($\overline{S}$ 4)	A1	D, R	C1.2, C1.3, C1.6, C1.8, C2.6, C2.7, C3.2, C4.1, C4.4, C4.5
BPEL-SPE (Kloppmann et al., 2005)	Yes	No ($\overline{S}$ 2)	A1	D, R	C1.1, C1.3, C1.4, C1.6, C1.8, C2.5, C2.7, C3.2, C4.1, C4.4, C4.5, C4.6
Business Rules Integration in BPEL (Rosenberg and Dustdar, 2005)	No	n/a	n/a	R	C1.2, C1.4, C1.8, C2.1, C3.2, C4.4, C4.5
Business Rules Integration (Oracle, 2006)	No	n/a	A3	R	C1.2, C1.4, C1.6, C2.1, C3.2, C4.1, C4.5
Collaborative Scopes (IBM, 2009)	Yes	Yes	A1	D, R	C1.3, C2.1, C3.1, C3.2, C4.1, C4.5
Continue on error (IBM, 2009)	Yes	Yes (s8)	A1	D,R	C1.7, C2.7, C3.1, C4.1, C4.5
Context4BPEL (Wieland et al., 2007)	Yes	Yes (s9)	A1	D, R	C1.3, C2.1, C2.2, C2.3, C2.7, C3.2, C4.1, C4.5, C4.6
Cross-process fault handling (Kopp et al., 2009)	No	No ($\overline{S}$ 4)	n/a	D, R	C1.3, C1.7, C2.1, C3.1, C3.2, C4.1, C4.5
Dedicated Administrator (IBM, 2009)	Yes	Yes (s8)	A1	D, R	C1.7, C2.5, C3.1, C4.1, C4.5
E4X extension for BPEL (van Lessen et al., 2009)	Yes	Yes (s7, s10)	A1	D, R	C1.4, C2.3, C3.2, C4.1, C4.5
Execution as Subprocess (IBM, 2009)	Yes	Yes (s8)	A1	D, R	C1.3, C2.1, C2.7, C3.1, C3.2, C4.1, C4.5
Extended WS-RM" (Charfi et al., 2009)	No	n/a	n/a	n/a	C1.3, C2.7, C3.1, C4.5
Generalized Flow (IBM, 2009)	Yes	No ($\overline{S}$ 4)	A1	D, R	C1.3, C1.8, C2.1, C3.2, C4.1, C4.5
Headers handling (Apache ODE group, 2009)	Yes	No ($\overline{S}$ 3)	A1	D, R	C1.8, C2.1, C3.2, C4.1, C4.5
id attribute	Yes	Yes	n/a	D	C1.4, C2.5, C3.2, C4.1
Ignore unavailable data (Apache ODE group, 2009)	Yes	No ($\overline{S}$ 3, $\overline{S}$ 4)	A1	D, R	C1.3, C1.4, C2.1, C3.2, C4.1, C4.5
ii4BPEL (IBM, 2006)	Yes	Yes (s7)	A1, A2, A3	D, R	C1.3, C1.5, C2.2, C3.1, C3.2, C4.1, C4.4, C4.5
Implicit correlations (Apache ODE group, 2009)	Yes	No ($\overline{S}$ 4)	A1	D, R	C1.3, C2.7, C3.1, C4.1, C4.5
Iterable forEach (Apache ODE group, 2009)	Yes	No ($\overline{S}$ 3, $\overline{S}$ 4)	A1	D, R	C1.3, C2.1, C3.2, C4.1, C4.5
Java Snippets (IBM, 2009)	Yes	Yes	A1	D, R	
"Retry or alternative service" (Juhnke et al., 2009 and Wen et al., 2006)	No	n/a		R	C1.7, C2.6, C2.7, C3.1, C4.4, C4.5
Microflows (IBM, 2009)	Yes	Yes	A1	D, R	C1.5, C2.5, C3.1, C4.1, C4.5
New XPath functions, e.g., Apache ODE group (2009)	No	Yes		D, R	C1.3, C2.5, C3.2, C4.1, C4.5

Extension	Extd	Conform	A	Type	Characteristics
Non-compensatable scopes (IBM, 2009)	Yes	No ($\overline{S}$ 4)	A1	D, R	C1.3, C2.5, C3.1, C4.1, C4.5
Oracle Human Task (Oracle, 2007)	Yes	Yes (s9)	A3	D	C1.8, C2.7, C3.1, C4.1
Oracle Notification Service (Oracle, 2007)	Yes	Yes (s9)	A3	D	C1.8, C2.7, C3.1, C4.1
"OWL for BPEL" (Le et al., 2009)	Yes	No ($\overline{S}$ 4)	A1	D, R	C1.2, C2.1, C3.1, C4.1, C4.5
Parameterized Processes (Karastoyanova, 2006)	Yes	No ($\overline{S}$ 2)	A1	D, R	C1.2, C1.6, C1.7, C2.6, C2.7, C3.2, C4.1, C4.4, C4.5, C4.6
Pluggable Framwork for Enabling the Execution of Extended BPEL Behavior (Khalaf et al., 2007)	No	n/a	n/a	R	C1.2, C2.1, C2.2, C3.1, C4.5
Process context (Apache ODE group, 2009)	Yes	No ($\overline{S}$ 4)	A1	D, R	C1.3, C1.4, C2.1, C2.2, C3.2, C4.1
References in BPEL (Wieland et al., 2009)	Yes	No ($\overline{S}$ 2)	A2, A3	D	C1.8, C2.2, C2.8, C2.4, C3.1, C3.2, C4.1, C4.4
Resource-oriented BPEL (Apache ODE group, 2009; Overdick, 2003)	Yes	No ($\overline{S}$ 4)	A1	D,R	
Retry Scopes (Eberle et al., 2009)	Yes	No ($\overline{S}$ 4)	A1	D, R	C1.3, C1.7, C2.1, C3.2, C4.1, C4.5
SH-BPEL (Modafferi & Mussi & Pernici, 2006)	No	n/a	n/a	R	C1.7, C2.7, C3.1, C4.5
SWRL for BPEL (Wu et al., 2008)	Yes	Yes (s9)	A2	D	C1.4, C2.1, C3.2, C4.1
T-BPEL (Tai et al, 2004)	Yes	Yes (s8)	A2	D, R	C1.3,C2.7,C3.1,C4.1,C4.5
Transaction boundaries (IBM, 2009)	Yes	No ($\overline{S}$ 4)	A1	D, R	C1.5, C2.5, C3.1, C4.1, C4.5
WS-BPEL extension for versioning (Juric et al., 2009)	Yes	No ($\overline{S}$ 2)	A1	D, R	C1.4, C2.5, C3.1
xBPEL (Chakraborty et al, 2004)	Yes	No ($\overline{S}$ 2)	A2, A3	D	C1.2, C1.3, C2.1, C3.3, C4.1, C4.5

8. Conclusions

BPEL extensions are omnipresent in research and industry, but no comparison or classification was available, neither there are best practices and recommendations for design and implementation of extensions. The only related research is a solution for architectural decision points (Zimmermann et al., 2009), but there are no decision points defined specially for BPEL extensions, yet. Balko et al. (2009) regard extensibility as a property of a process model to be adaptable. This is in contrast to our definition, which regards the extensibility of the modeling language itself.

The main contribution of this paper is a comprehensive framework for understanding and classifying BPEL extensions, and a recommendation for developing BPEL extensions properly. For providing that knowledge, first the classification for BPEL extensions is given and based on that an overview of the state of the art of BPEL extensions is given. Furthermore as practical advice we give a design guideline that raises different questions for deciding wether a BPEL extension has to be implemented or the functionality can be realized in another way.

Interesting to note is that only around half of the discussed extensions are standard-conform BPEL extensions in terms of Definition 1. Standard-conform extensions have their advantage in being portable and re-usable across different BPEL environments. Needless to say, non-conforming extensions also have their justification. Thus, if an extension is not conforming to the BPEL standard, it does not imply that it is of no use or that it is realized in a wrong way. As we have shown, valid extensions to BPEL can include anything ranging from new attributes to new elements, to extended assign operations up to completely new activities. We have also shown that missing functionality can be implemented in different ways, for instance using standard language constructs or introducing extension attributes or extension activities. However, when talking about extensions, one has to be aware that the ways of extending BPEL foreseen in the specification are limited.

The presented discussion on possibilities to realize an extension remains valid in context of the Business Process Model and Notation (BPMN) language. As part of our future work, we will classify BPMN extensions according to the presented classification framework.

Acknowledgements

The work published in this paper was partially funded by the COMPAS project (contract no. FP7-215175) and the ALLOW project (contract no. FP7-213339) under the EU 7th Framework Programme Information and Communication Technologies Objective, the DFG Cluster of Excellence Simulation Technology (EXC310), the DFG project Nexus (SFB627), and the S-Cube project under the Network of Excellence (contract no. FP7-215483).

References

van der Aalst, W.M.P. & Weske, M. & Grünbauer, D. (2004), Case handling: a new paradigm for business process support, 53(2), *Data & Knowledge Engineering*, Elsevier.

Adams, P. & Easton, P. & Johnson, Eric & Merrick, R. & Phillips, M. (2010). SOAP over Java Message Service 1.0, W3C Working Draft 26 October 2010.

Agrawal, A. & Amend, M. & Das, M. & Ford, M. & Keller, C. & Kloppmann, M. & König, D. & Leymann, F. & Müller, R. & Pfau, G. & Plösser, K. & Rangaswamy, R. & Rickayzen, A. & Rowley, M. & Schmidt, P. & Trickovic, I. & Yiu, A. & Zeller M. (2007a). WS-BPEL Extension for People (BPEL4People), Version 1.0, White Paper.

Agrawal, A. & Amend, M. & Das, M. & Ford, M. & Keller, C. & Kloppmann, M. & König, D. & Leymann, F. & Müller, R. & Pfau, G. & Plösser, K. & Rangaswamy, R. & Rickayzen, A. & Rowley, M. & Schmidt, P. & Trickovic, I. & Yiu, A. & Zeller M. (2007b). Web Services Human Task (WS-HumanTask), Version 1.0, White Paper.

Alonso, G. & Casati F. & Kuno, H. & Machiraju, V. (2004), Web Services, Springer.

Apache ODE group (2009). BPEL Extensions. URL: `http://ode.apache.org/bpel-extensions.html`

Balko, S. & ter Hofstede, A. H. M. & Barros, A. P. & Rosa, M. (2009). Controlled Flexibility and Lifecycle Management of Business Processes through Extensibility. *3rd International Workshop on Enterprise Modelling and Information Systems Architectures*, GI.

Barros, A. & Decker, G. & Dumas, M. & Weber, F. (2007). Correlation Patterns in Service-Oriented Architectures. Proceedings of the 9th International Conference on Fundamental Approaches to Software Engineering (FASE), Springer Verlag.

Bischof, M & Kopp, O. & van Lessen, T. & Leymann, F. (2009). BPELscript: A Simplified Script Syntax for WS-BPEL 2.0. *35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009).*

Blow, M. & Goland, Y. & Kloppmann, M. & Leymann, F. & Pfau, G. & Roller, D. & Rowley, M. (2004). BPELJ: BPEL for Java, White Paper.

Chakraborty, D. & Lei, H. (2004) Pervasive Enablement of Business Processes, *Proc. of the Second IEEE Intl. Conf. on Pervasive Computing and Communications (PerCom 2004).*

Chappell, D. (2004). Enterprise Service Bus: Theory in Practice, O'Reilly Media.

Charfi, A. & Mezini, M. (2004). Aspect-Oriented Web Service Composition with AO4BPEL. In: *European Conference on Web Services (ECOWS)*.

Charfi, A. & Schmeling, B. & Mezini, M. (2006). Reliable Messaging for BPEL Processes. *International Conference on Web Services (ICWS)*. IEEE.

Christensen, E. & Curbera, F. & Meredith, G. & Weerawarana, S. (2001). Web Services Description Language (WSDL) 1.1.

Curbera, F. & Duftler, M. & Khalaf, R. & Lovell, D. Bite: Workflow Composition for the Web. *International Conference on Service Oriented Computing (ICSOC)*, Springer

Decker, G. & Kopp, O. & Leymann, F. & Weske, M. (2009). Interacting services: from specification to execution, *Data & Knowledge Engineering*, doi:10.1016/j.datak.2009.04.003.

Dörnemann, T. & Friese, T. & Herdt, S. & Juhnke, E. & Freisleben B. (2007). Grid Workflow Modelling Using Grid-Specific BPEL Extensions, *Proceedings of German e-Science Conference 2007*, pp. 1-9.

Eberle, H. & Kopp, O. & Unger, T. & Leymann. (2009). F. Retry Scopes to Enable Robust Workflow Execution in Pervasive Environments, *2nd Workshop on Monitoring, Adaptation and Beyond (MONA+)*.

Fonden, C. (2009): Konzeption und Entwicklung von Kontexterweiterungen für Workflows, Diploma Thesis 2901, Institute of Architecture of Application Systems, University of Stuttgart.

Habich, D., Richly, S., Preissler, S., Grasselt, M., Lehner, W., Maier, A. (2007) BPEL-DT - Data-aware Extension of BPEL to Support Data-Intensive Service Applications. In C. Pautasso and T. Gschwind, editors, *WEWST*, volume 313 of CEUR Workshop Proceedings. CEUR-WS.org.

Hackmann, G. & Gill, C. & Roman, G.-C. (2007). Extending BPEL for Interoperable Pervasive Computing. IEEE International Conference on Pervasive Computing.

IBM (2006). Adding an information service activity to a business process, URL: http://publib.boulder.ibm.com/infocenter/dmndhelp/v6rxmx/index.jsp?topic=/com.ibm.is.bpel.help.doc/topics/accessdata.htm

IBM (2007). Mapping Specification Language. http://www.research.ibm.com/journal/sj/452/roth.html

IBM (2009). Working with BPEL extensions, URL: http://publib.boulder.ibm.com/infocenter/dmndhelp/v6r2mx/index.jsp?topic=/com.ibm.wbit.620.help.bpel.ui.doc/concepts/cextent.html.

International Organization for Standardization (2006). Information Technology — ECMAScript for XML (E4X) Specification. ISO/IEC 22537:2006

Jäger, C. & Wolke, C. (2008) Make-or-Buy Decisions - A Transaction Cost Theoretical Approach to the Assessment of Outsourcing Activities, Books on Demand GmbH.

Juhnke, E. & Dörnemann, T. & Freisleben, B. (2009). Fault-Tolerant BPEL Workflow Execution via Cloud-Aware Recovery Policies. *Proceedings of the 35th EUROMICRO Conference on Software Engineering and Advanced Applications*.

Juric, M. & Sasa, A. & Rozman I. (2009). WS-BPEL Extensions for Versioning, *Information and Software Technology 51*, pp. 1261–1274.

Karastoyanova, D. (2006). Enhancing Flexibility and Reusability of Web Service Flows through Parameterization, *PhD thesis*, TU Darmstadt and Universität Stuttgart.

Karastoyanova, D. & Khalaf, R. & Schroth, R. & Paluszek, M. & Leymann, F. (2006). BPEL Event Model. *University of Stuttgart, Technical Report Computer Science No. 2006/10.*

Karastoyanova, D. & Leymann, F. (2009). BPEL'n'Aspects: Adapting Service Orchestration Logic, *Proceedings of 7th IEEE International Conference on Web Services (ICWS)*.

Kareliotis, C. & Vassilakis, C. & Georgiadis, P. (2007). Enhancing BPEL scenarios with Dynamic Relevance-Based Exception Handling. In: *ICWS*, 751-758

Khalaf, R. (2008). Supporting business process fragmentation while maintaining operational semantics: a BPEL perspective, Dissertation, University of Stuttgart, Germany.

Khalaf, R. & Karastoyanova, D. & Leymann, F. (2007). Pluggable Framework for Enabling the Execution of Extended BPEL Behavior. *Proceedings of the 3rd International Workshop on Engineering Service-Oriented Application (WESOA'2007)*.

Khalaf, R. & Leymann, F. (2006). Role-based Decomposition of Business Processes using BPEL. *International Conference on Web Services (ICWS)* (pp. 770-780).

Khalaf, R. & Subramanian, R. & Mikalsen, T. & Duftler, M. & Diament, J. & Silva-Lepe, I. Enabling Community Participation for Workflows through Extensibility and Sharing, *Workshop on Business Process Management and Social Software (BPMS2'09)*, Springer.

Kiczales, G. (1997). Aspect-Oriented Programming, *Proceedings of ECOOP'97*.

Kloppmann, M. & Koenig, D. & Leymann, F. & Pfau, G. & Rickayzen, A. & Riegen, C. & Schmidt, P. & Trickovic, I. (2005). WS-BPEL Extension for Sub-processes - BPEL-SPE, *White Paper*.

Kloppmann, M. & König, D. & Leymann, F. & Pfau, G. & Rickayzen, A. & von Riegen, C. & Schmidt, P. & Trickovic, I. (2005). WS-BPEL Extension for People – BPEL4People, *White Paper*.

König, D. & Lohmann, N. & Moser, S. & Stahl, C. & Wolf, K. (2008): Extending the compatibility notion for abstract WS-BPEL processes. *WWW 2008*, 785-794

Kopp, O. & Martin, D. & Wutke, D. & Leymann, F. (2009). The Difference Between Graph-Based and Block-Structured Business Process Modelling Languages. *Enterprise Modelling and Information Systems. Vol. 4(1)*, pp.3-13.

Kopp, O. & Wieland, M. & Leymann, F. (2009). Towards Choreography Transactions. *1st Central-European Workshop on Services and their Composition (ZEUS)*.

Laemmel, R. & Ostermann, K. (2006). Software Extension and Integration with Type Classes, *Proceedings of the 5th international conference on Generative Programming and Component Engineering (GPCE '06)*.

Lau, C., Beaton, M. (2004) Architecting on demand solutions, Part 3: Use BPEL to create business processes, *IBM developerWorks*. URL: `http://www.ibm.com/developerworks/library/i-odoebp3/`

Le, Duy Ngan & Nguen, Ngoc Son & Mous, Karel & Ko, Ryan Kok Leong & Goh, Angela Eck Soong. (2009). Generating Request Web Services from Annotated BPEL. *RIVF International Conference on Computing and Communication Technologies*.

van Lessen, T. & Leymann, F. & Mietzner, R. & Nitzsche, J. & Schleicher, D. (2009). A Management Framework for WS-BPEL. *Proceedings of the 6th IEEE European Conference on Web Services*.

van Lessen, T. & Nitzsche, J. & Karastoyanova, D. (2009). Facilitating Rich Data Manipulation in BPEL using E4X. . *1st Central-European Workshop on Services and their Composition (ZEUS)*.

Leymann, F. (2005). The (Service) Bus: Services Penetrate Everyday Life. *International Conference on Service-Oriented Computing (ICSOC).*

Leymann, F. & Roller, D. (2000). Production workflow: concepts and techniques, *Prentice Hall PTR*.

Leymann, F. & Roller, D. (2006). Modeling business processes with BPEL4WS. *Information Systems and e-Business Management (ISeB)*, Springer. 265-284.

Liu, A. & Li, Q. & Huang, L. & Xiao, M. (2007). A Declarative Approach to Enhancing the Reliability of BPEL Processes, *IEEE International Conference on Web Services (ICWS)*.

Ma, Z. & Leymann, F. (2009), BPEL Fragments for Modularized Reuse in Modeling BPEL Process. *5th International Conference on Networking and Services (ICNS).*

Modafferi, S. & Conforti, E. (2006). Methods for enabling recovery actions in ws-bpel. *In Proc. of Int. Conf. on Cooperative Information Systems (CoopIS)*.

Modafferi, S. & Mussi, E. & Pernici, B. (2006). SH-BPEL: a self-healing plug-in for Ws-BPEL engines. *Proceedings of the 1st Workshop on Middleware for Service Oriented Computing, MW4SOC*, ACM.

Nitzsche, J. & van Lessen, T. & Karastoyanova, D. & Leymann, F. (2007a). BPEL^{light}, *Proceedings of the 5th Interational Conference on Business Process Management (BPM)*.

Nitzsche, J. & van Lessen, T. & Karastoyanova, D. & Leymann, F. (2007b). BPEL for Semantic Web Services (BPEL4SWS). In: Proceedings of the 3rd International Workshop on Agents and Web Services in Distributed Environments (AWeSome'07) - On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops. Lecture Notes in Computer Science; 4805/2007, Springer.

OASIS (2004). Web Services Reliable Messaging TC: WS-Reliability 1.1, *OASIS Standard*. http://docs.oasis-open.org/wsrm/ws-reliability/v1.1.

OASIS (2007). Web Services Business Process Execution Language Version 2.0, *OASIS Standard*.

Oracle (2006). BPEL Process Manager: BPEL + Business Rules. URL: `http://www.oracle.com/technology/products/ias/bpel/pdf/bpelandbusinessrules.pdf`

Oracle (2007). BPEL Process Manager Developer’s Guide. Version 10g (10.1.3.1.0) B28981-03. URL: `http://download.oracle.com/docs/cd/B31017_01/integrate.1013/b28981.pdf`

Overdick, H. (2003). Towards resource-oriented BPEL. *2nd ECOWS Workshop on Emerging Web Services Technology*.

Pautasso, C. (2008). BPEL for REST. 7^{th} *International Conference on Business Process Management*.

Peltz, C. (2003). Web Services Orchestration and Choreography, *IEEE Computer*, 36, 46-52.

Rosenberg, F. & Dustdar, S. (2005). Business Rule Integration in BPEL – A Service-Oriented Approach, Proceedings of the 7th International IEEE Conference on E-Commerce Technology.

Schumm, D. (2007) A Graphical Tool for Modeling BPEL 2.0 Processes, *Universität Stuttgart, Fakultät Informatik, Studienarbeit Nr. 2124*.

Schumm, D. & Karastoyanova, D. & Leymann, F. & Nitzsche, J. (2009). On Visualizing and Modelling BPEL with BPMN, *Proceedings of the 4th International Workshop on Workflow Management (ICWM)*.

Silva-Lepe, I. & Subramanian, R. & Rouvellou, I. & Mikalsen, T. & Diament, J. & Iyen-gar, A. (2008). SOAlive Service Catalog:ASimplied Approach to Describing, Discovering and Composing Situational Enterprise Services. *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*.

Stahl, T. & Völter, M. & Czarnecki, K. (2006). Model-driven Software Development*: technology, engineering, management*. John Wiley & Sons.

Stein, S. & Kühne, S. & Ivanov, K. (2009). Business to IT Transformations Revisited. Springer-Verlag

Tai, S. & Mikalsen, T. A. & Wohlstadter, E. & Desai, N. & Rouvellou, I. (2004). Transaction policies for service-oriented computing, *Data Knowl. Eng*, 51, 59-79.

Vedamuthu et. al (2007a). Web Services Policy 1.5 – Framework, *W3C Recommendation 04 September 2007*, URL: `http://www.w3.org/TR/ws-policy/`.

Vedamuthu et al. (2007b). Web Services Policy 1.5 – Attachment, W3C Recommendation 04 September 2007, URL: `http://www.w3.org/TR/ws-policy-attach/`.

Vrhovnik, M. & Schwarz, H. & Suhre, O. & Mitschang, B. & Markl, V. & Maier, A. & Kraft T. (2007). An approach to optimize data processing in business processes, *Proceedings of the 33rd international conference on Very large data bases* 2007, pp. 615-626.

Vrhovnik, M. & Schwarz, H. & Radeschiitz, S. & Mitschang B. (2008). An Overview of SQL Support in Workflow Products, *IEEE 24th International Conference on Data Engineering*.

Weidlich, M. & Decker, G. & Großkopf, A. & Weske. M. (2008). BPEL to BPMN: The Myth of a Straight-Forward Mapping. *International Conference on Cooperative Information Systems (CoopIS)*.

Weerawarana, S. & Curbera, F. & Leymann, F. & Storey, T. & Ferguson, D.F. (2005). Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More, *Prentice Hall PTR*.

Wen, Jiajia & Chen, Junlinag & Peng, Yong & Xu, Meng. (2006). A Multi-Policy Exception Handling System for BPEL Processes. *First International Conference on Communications and Networking in China*.

Wieland, M. & Görlach, K. & Schumm, D. & Leymann, F. (2009). Towards Reference Passing in Web Service and Workflow-based Applications, *Proceedings of the 13th IEEE Enterprise Distributed Object Conference (EDOC 2009)*.

Wieland, M. & Kopp, O. & Nicklas, D. & Leymann, F. (2007). Towards Context-aware Workflows, *CAiSE'07 Proceedings of the Workshops and Doctoral Consortium Vol 2*.

Wu, Yunzhou & Dohsi, Prashant. (2008). Making BPEL Flexible – Adapting in the Context of Coordination Constraints Using WS-BPEL, *International Conference on Services Computing (SCC 2008).*

Zhang, Huajian & Fan, Xiaoliang & Zhang, Ruisheng & Lin, Jiazao & Zhao, Zhili & Li, Lian (2008). Extending BPEL2.0 for Grid-Based Scientific Workflow Systems, *Asia-Pacific Services Computing Conference (APSCC '08).*

Zimmermann, O. & Koehler, J. & Leymann, F. & Polley, R. & Schuster N. (2009). Managing Architectural Decision Models with Dependency Relations, Integrity Constraints, and Production Rules. *Journal of Systems and Software, Elsevier.* 82(8), pages 1249-1267.