Dynamic Device and Service Discovery Extensions for WS-BPEL

Hendrik Bohn, Frank Golatowski, *Member, IEEE,* and Dirk Timmermann, *Member, IEEE*

***Abstract*—Process management systems reveal the full potential of Service Oriented Architectures by composing heterogeneous hardware and software components (exposed as services) to powerful distributed applications. For Web services architectures, the Web Services Business Process Execution Language (WS-BPEL) is the standard for centralised Web service orchestration which is governed by OASIS. Unfortunately, WS-BPEL is quite static and addresses dynamically changing environments insufficiently as devices and services might change their addresses or be replaced by others. This paper presents extensions to WS-BPEL in order to discover devices and services as introduced by WS-Discovery. It also describes a discovery service for WS-BPEL processors not supporting presented extensions.**

***Index Terms*—Web services discovery, Web services orchestration, WS-BPEL, DPWS.**

I. Introduction

DISTRIBUTED applications integrating hardware and software components have strong requirements on the underlying software architectures. The participating hard- and software components are often very heterogeneous and from different manufacturers. They might wander between networks, enter and leave networks dynamically or might be replaced by similar components taking over their tasks. Distributed applications might not be aware of the specific components being used at run-time for certain tasks. The mentioned aspects do not only require a homogeneous view on hard- and software components and standardized communication protocols but also means for automation and composition of components.

The paradigm of *Service Oriented Architectures* (SOA) [1] already represents a step forward by exposing the functionality of hard- and software components as services. Service interfaces are standardised and self-describable. SOAs provide means for the self-description, announcement, discovery, interaction and usage of services. The SOA implementation with the highest market penetration today is Web services. *Web services* [2] are a set of protocol building blocks which can be composed in various ways to suit a certain purpose. A primary purpose is called *profile* and defines a specific subset of Web service protocols including required adaptations. For example, the *Devices Profile for Web Services* (DPWS) [3] consists of Web service protocols to enable addressing (WS-Addressing), secure messaging (WS-Security), discovery (WS-Discovery), description (WSDL) and eventing (WS-Eventing) on resource-constrained devices and their hosted services.

Complex interactions between services and service clients can be modelled as workflows or processes. The planning, modelling, execution and controlling of Web service processes can be performed using the *Web Services Business Process Execution Language* (WS-BPEL) [4] and corresponding tools. WS-BPEL is an XML format to describe machine-readable processes which is based on Web services standards. It is governed by the Organization for the Advancement of Structured Information Standards (OASIS) and currently available in version 2.0. WS-BPEL processes can be designed using graphical tools and can be executed on a WS-BPEL engine. WS-BPEL supports the *Basic Profile* [5] which provides interoperability guidance for the Web service core specifications such as Web services communication protocols (SOAP), description (WSDL), registry (UDDI) and also WS-Addressing in the upcoming version. Unfortunately, these limitations insufficiently address processes in dynamical environments and the integration of devices. Participating Web services have to be known during the process design phase, mobile services are not considered and discovery of devices and services is not supported (as promoted by DPWS).

This paper discusses extensions of WS-BPEL to support the dynamic discovery of devices and services as promoted by DPWS. Thereby, it is not required to know the exact service and corresponding device at process design time. Furthermore, the paper proposes a discovery service which can be used by any standard WS-BPEL process for dynamic discovery of devices and services.

The paper is organised as follows: Section II presents an overview of the fundamentals for the discovery approach. The design of a discovery extension for WS-BPEL are explained in section III as well as the concept for a dedicated discovery service being useful for legacy WS-BPEL engines. The implementation of presented approach is described in section IV. A summary and future development is presented in section V.

II. Fundamentals and related work

A. WS-BPEL

The *Web Services Business Process Execution Language* (WS-BPEL) [4] is an executable XML language for the description of Web service processes. WS-BPEL distinguishes between abstract processes and concrete processes. Whereas concrete processes are directly executable on a WS-BPEL engine and processor, respectively, abstract processes enable developers to describe the general message exchanges between

This work has been achieved in the European ITEA project LOMS (www.loms-itea.org) and has been funded by German Federal Ministry of Education and Research under contract number 01—SF11H.

H. Bohn, F. Golatowski, and D. Timmermann are with the Institute of Applied Microelectronics and Computer Engineering, University of Rostock, 18051 Rostock, Germany (e-mail: hendrik.bohn, frank.golatowski, dirk.timmermann@uni-rostock.de).

Manuscript received December 10, 2007.

Web services and omit concrete behaviour. The current version WS-BPEL 2.0 is based on XML Schema for describing data structures, XPath for getting access to all elements in an XML structure and WSDL as self-described interfaces to participating Web services. A WS-BPEL process can be modelled using graphical tools. It is deployed as a Web service on a WS-BPEL engine. Therefore, WS-BPEL processes possess their own WSDL description and are executed by being invoked from outside. Every invocation of a process Web service starts a new instance of the WS-BPEL process on an engine. WS-BPEL provides several concepts in order to support flexible processes which are introduced below.

Web service interaction: A step in the performance of a WS-BPEL process is called *activity*. WS-BPEL interacts with other Web services using activities for receiving data from inbound invocation starting the process instance, for replying to it and for invoking other Web services. Interactions with external Web services are represented by *partner links*. Synchronous and asynchronous Web service interactions are supported and specified in *partner link types*. *Callback* interfaces to the process are used by external Web services for asynchronous interactions.

Manipulation of data: WS-BPEL allows the extraction of data from messages and XML Schema using XPath and also supports XSL transformations on XML structures. Data can be copied and assigned to variables as well as validated against XML Schema declarations.

Structuring process: WS-BPEL uses validity scopes to distinguish between local and global contexts. Each scope can define its own fault and compensation handler and local variables. Activities can be processed in loops or selected according to certain conditions. Parallel as well as sequential activities are also supported. Dependencies of activities can be expressed using *links* (specified by `<sources>` and `<targets>`). An activity being dependent on others is only performed if the dependency condition evaluates to true (e.g. activity C is dependent on the performance of A and also B, C is only started if both A and B have been performed).

Correlation: In order to relate a certain message to a specific instance, WS-BPEL introduces the concept of correlation. It allows the identification of a certain context common to all messages of one instance.

Event, exception and compensation handling: Messages received by a process or time-outs can be handled in parallel to the execution of the process. WS-BPEL also supports mechanisms for exception handling if a fault is thrown during the execution of a process. In case several activities are completed and an occurring error requires undoing completed activities, compensation handling can be used for it. This is important as participating Web services are normally stateless in a process.

Extensibility: WS-BPEL provides means for adding new activities and data assignments operations.

B. DPWS

The *Devices Profile for Web Services* (DPWS) was developed to enable secure Web service capabilities on resource-constraint devices [3]. It is part of the current Microsoft

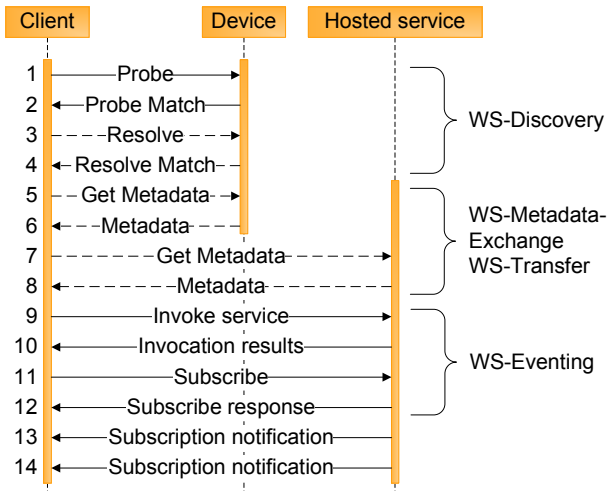

Fig. 1. The lifecycle of a DPWS devices and its hosted services.

Windows version Vista and bases on the requirements that devices may change their networks as they are mobile (discovery), might have limited resources (small Web services stack), and that services are *hosted services* as they reside on a specific devices (location based). DPWS consists of several Web service protocols and their adaptations. It takes care of addressing of devices and hosted services (WS-Addressing), discovering devices (WS-Discovery), exchanging meta data about devices and hosted services (WS-MetadataExchange, WS-Transfer), providing a publish/subscribe mechanism for state changes on hosted services (WS-Eventing), exchanging requirements on the usage of devices and services (WS-Policy) as well as enabling secure messaging (WS-Security).

The lifecycle of a device and its hosted services is shown in figure 1. The client sends a multicast `Probe` message specifying scope and/or type of a desired service (*target service*) as described by WS-Discovery [6] (message 1). The semantics behind scopes and types are not specified by WS-Discovery but scopes can be used to identify the location scope whereas types could identify service classes. The `Probe` message also indicates if the client wishes to use secure messaging (WS-Security). All devices listen for `Probe` messages. In case the scope and/or target service matches one of its hosted services the device responds with a `ProbeMatch` message (message 2). The `ProbeMatch` message contains the device's Endpoint Reference as defined in WS-Addressing, supported transport protocols and security requirements and capabilities. *Endpoint references* (EPR) are XML structures including destination address (destination Web service is called endpoint), and optional meta data describing the service (e.g. usage requirements). If security is desired, the client sets up a secure channel in an additional message. In case the endpoint reference does not include a physical address, the client can use a `Resolve` message to retrieve it from the device in a `ResolveMatch` message (message 3 and 4). In message 5, the client can directly request more information about the device using a `GetMetadata` message using WS-Transfer. The desired device will respond either with included meta data (format defined by WS-MetadataExchange) or by providing a

reference to the meta data (message 6). The meta data includes device details and endpoint references to each hosted service. Similar to message 5, the client requests more information about the desired hosted service in message 7. The desired hosted service sends its meta data to the client including its WSDL description (message 8). The *Web Service Definition Language* (WSDL) is a standard for describing abstract and concrete interfaces to Web services including operations and transmitted messages. Hosted services are invoked by receiving an invocation message from the client according to corresponding operation described in the WSDL (message 9 and 10). Message 11 to 14 illustrates the publish/subscribe functionality defined by WS-Eventing. Subscribers can get informed about state changes related to specific actions of a Web service.

The *Web Services for Devices* (WS4D) initiative [7] was founded by the University of Rostock and the University of Dortmund (with support of MATERNA GmbH and Schneider Electric, France) in order to promote the use of DPWS and its further development. As this paper contributes to the WS4D initiative, the namespace `ws4d` will be used to identify the particular extensions to WS-BPEL.

C. Related work

There are several approaches for providing discovery functionality, among them are Web service registries, Web service search engines, dedicated discovery Web services and other Web service discovery approaches.

The best-known Web service registry is *Universal Description, Discovery and Integration* (UDDI) [8] which is part of the WS-I Basic Profile and governed by OASIS. Web services can register itself at the UDDI server and clients request the UDDI server for available services. Beside the fact that the support for UDDI in commercial applications is declining, UDDI has to be set up as a dedicated application in a network and supports mainly static Web services with long-term availability. The support for Web services on devices is very limited and interaction with WS-BPEL processes is not provided.

A number of search engines usable for Web service discovery have been evaluated by Bachlehner et al. [9]. Specialised Web service search engines (e.g. XMethods, Woogle) either crawl the web or access databases of manually registered services. Standard Web search engines (e.g. Google, Baidu) can also be used by restricting their results to WSDL files. Unfortunately, Web service search engines have the same limitations as UDDI in terms of discovery of devices and WS-BPEL support.

Numerous approaches make use of dedicated discovery services which can be invoked by Web service clients or WS-BPEL processes as propagated by the Web Services Architecture (WSA) [2]. They focus on semantic discovery aspects, discovery in grids and/or discovery for service composition. An elaborated overview of current Web service discovery mechanisms is presented by Garofalakis et al. [10]. WS-BPEL supports the use of dedicated discovery services by a mechanism for dynamical binding of endpoint references.

```
<extensions>
  <extension namespace="http://www.ws4d.org/bpeldiscovery"
    mustUnderstand="yes" />
</extensions>
```

```
<extensionActivity>
  <ws4d:discoverDev devRef="EPR-list" standard-attributes>
    standard-elements
  </ws4d:discoverDev>
</extensionActivity>
```

Fig. 2. WS-BPEL extension declaration for discovery.

Unfortunately, all approaches do not support device discovery and do not comply with the requirements of DPWS. Therefore, the recent device discovery protocol WS-Discovery was developed which will be used in this work in conjunction with WS-BPEL.

III. Design of the discovery process

A. Concept for device and service discovery in WS-BPEL

The device and service discovery proposed for WS-BPEL is oriented at the approach as proposed by DPWS (shown in figure 1). It is based on the Web service protocols WS-Discovery, WS-MetadataExchange and WS-Transfer which have to be provided by the WS-BPEL engine in order to support device and service discovery.

The device and service discovery used for WS-BPEL involves two stages. In the first stage, the devices are discovered and their endpoint references are returned. The second stage is the discovery of desired hosted services related to a device's endpoint reference. The dynamic endpoint reference capability of WS-BPEL (`<sref:service-ref>`) is used to dynamically set the endpoint references for devices and services inside the process. In order to use a service which will be discovered during the execution of a WS-BPEL process, its abstract WSDL description must be present. These WSDLs are defined by the developer of the process and represent desired services.

Legacy WS-BPEL engines which do not offer support for proposed discovery mechanisms can invoke a dedicated discovery service. The discovery service (proposed in this paper) has just the same functionality as the discovery extension to WS-BPEL.

All concepts are described below in more detail.

B. Wild-card WSDL

All services being discovered during the execution of a process must be described in a separate WSDL at design-time. These WSDLs include all abstract information about the desired services including messages, operations, variables and partner link types. As introduced above, partner link types represent the interactions between a process and external Web services. All concrete information such as binding to underlying protocols and endpoint references are assigned during the discovery.

```
<ws4d:discoverDev scopes="QName-list"? types="QName-list"?
  devRef="EPR-list" />

<ws4d:discoverSvc scopes="URI-list"? types="QName-list"?
  devRef="EPR-list"? devAds="QName-list"? checkRef="EPR-
  list"? svcRef="EPR-list" partnerLink="QName"? />

<ws4d:validateSvc svcRef="EPR" partnerLink="QName" />
```

Fig. 3. WS-BPEL extension activities for synchronous discovery: Discovering devices, services and validating a specific service.

C. Extending WS-BPEL by discovery related activities

Figure 2 shows the use of extensions in WS-BPEL. An extension must be declared using the `<extension>` element including its namespace and information whether this extensions must be understood by the WS-BPEL engine. All extensions are wrapped into the element `<extensions>`. We use the namespace `http://www.ws4d.org/bpeldiscovery` (or shortly `ws4d`) for the extensions and request conformance to the extensions by the underlying engine.

Each additional activity is embedded into one `<extensionActivity>` element. Extension activities must support the standard attributes `name` and `suppressJoinFailure` as well as the standard elements. The attribute `name` specifies a name for the activity (default is unnamed) and `suppressJoinFailure` is used to suppress throwing a fault whenever a dependency condition between activities evaluates to false (`bpel:joinFailure`). The default value is the value from its closest enclosing activity. Standard elements are `<sources>` and `<targets>` of links (dependency relations between activities).

Three activities are defined for the discovery using WS-BPEL as shown in figure 3: Discovering one or more devices, one or more services, and checking the validity of discovered services for their use in the process. It may appear to be strange to have a separate activity for device discovery as WS-BPEL addresses composition of Web services only. However, this offers more flexibility for developers if they wish to apply additional selection rules depending on the devices being discovered. The new activities are defined as XML Schema under http://www.ws4d.org/schemas/bpeldiscovery.xsd using the namespace `http://www.ws4d.org/bpeldiscovery` and are described in detail below. Optional attributes are identified by a question mark.

Discovering devices

The device discovery activity is called `<ws4d:discoverDev>`. The optional attributes represent lists of scopes and types which can be used to constrain the search. If no scopes and types are assigned the search is performed for all discoverable devices. The attribute `devRef` (devices references) is mandatory and returns desired devices as a list of endpoint references. If no device is found which matches the search criteria, the fault `ws4d:noDeviceFound` is thrown.

```
<ws4d:discoverSvc />
<receive partnerLink="..."
  portType="ws4d:svcDiscoveryCallbackPT"
  operation="svcDiscoveryCallback" variable="svcEPR" />
```

Fig. 4. Asynchronous service discovery proposed for WS-BPEL.

Discovering services

The service discovery activity `<ws4d:discoverSvc>` enables developers to search for a specific service. We introduce three ways for searching for specific services: Checking the availability of a service, searching for a desired service on a number of specific devices or on all devices in reach. All services matching the search criteria are returned as a list of endpoint references by the attribute `svcRef` (service references). If no corresponding service is found an empty list is returned and the fault `ws4d:noServiceFound` is thrown.

The availability of one or more services is checked by specifying their endpoint reference in the `checkRef` attribute and evaluating these against returned service endpoint references. Services on specific devices can be discovered by specifying a list of device endpoint references (`devRef` attribute) and/or a list of physical device addresses (`devAds`). We recommend to use device endpoint references only as mobile devices might wander between networks and their physical device addresses would change accordingly. The discovery of services on all devices in reach is a composition of the `<ws4d:discoverDev>` activity with searching for a desired service on a list of devices (which are returned by the device discovery). Device scopes and types can be used as described in the previous subsection. Additionally, the attribute `partnerLink` can be used to fine-tune all three service discovery possibilities as the search results will also be checked for compliance to the operation, variables and message exchange pattern described in the WSDL under the corresponding partner link.

Rules for the use of attributes in the service discovery are as follows: All attributes except `svcRef` are optional. The selection criteria for returned service endpoint references is the intersection set of all provided (input) attributes. If the attributes `devRef` and/or `devAds` are provided, no explicit device discovery will be performed. Implicit device discovery is performed by requesting all service endpoint references from given devices as unavailable devices will not return anything. These rules have to be checked statically by the WS-BPEL engine when the process is deployed. A violating process definition must not be executed.

Validating services

The activity `validateSvc` can be used in case a service endpoint reference (`svcRef`) must be checked against an operation, associated variables or its message exchange pattern (request/response or one-way and asynchronous/synchronous interaction, respectively) represented by the `partnerLink` attribute. If the service endpoint reference evaluates as invalid

```
<ws4d:discoverSvc />
  <catch faultName="ws4d:noDeviceFound">
    activity
  </catch>
  <compensationHandler>
    activity
  </compensationHandler>
</ws4d:discoverSvc>
```

Fig. 5. Service discovery with fault and compensation handlers defined inline.

the fault `ws4d:invalidServiceRef` is thrown.

Asynchronous discovery

Above, we have described synchronous discovery where the process waits for the discovery results and proceeds then. In case the process is not required to wait for the results and they are only needed at a later stage of the process, asynchronous discovery can be used. This is also useful if the results of some devices are expected to be returned with a longer delay.

The use of asynchronous service and device discovery in WS-BPEL is shown in figure 4. It is indicated by omitting the attributes `devRef` in `<ws4d:discoverDev>` and `svcRef` in `<ws4d:discoverSvc>` which are mandatory for synchronous discovery. In order to receive the results, a callback functionality is provided by a `<receive>` activity following the asynchronous discovery activity. This is very similar to the asynchronous Web service invocation mechanism specified by WS-BPEL. The callback activity `<receive>` can also be used in an event handler of a process being processed in parallel to the actual process.

Asynchronous discovery requires the specification of a callback Web service interface (`portType`). We propose at least two callback interfaces. One for device and another for service discovery to avoid additional validation whether the incoming endpoint references belong to a device or service. The separation between callback interfaces according to other criteria is left to the developer.

Fault and compensation handling

The discovery faults `ws4d:invalidServiceRef`, `ws4d:noServiceFound` and `ws4d:noDeviceFound` must be handled by a fault handler. Fault handlers can be defined for an enclosing scope or inline. Figure 5 shows an example for the declaration of an inline fault handler similar to inline fault handling used for the Web service invocation activity. The process designer must specify the activities which are performed in case of a fault. Instead of catching a specific fault through `<catch>`, `<catchAll>` can also be used. `catchAll` handles all faults including dependency violations between activities (`bpel:joinFailure`). If no fault handler is declared in a process an occurring fault will cause an immediate termination of the process.

Compensation handlers can also be defined in an enclosing scope or inline. As specified by WS-BPEL activities defined in compensation handlers are only performed when the activities in their scope are performed successfully and these compensation handlers are called at a later stage in the process (e.g. due to a fault).

```
<xs:element name="DiscReq" type="DiscReqType" />
<xs:complexType name="DiscReqType">
  <xs:sequence>
    <xs:element name="Scopes" type="ws4d:URI-list"
      minOccurs="0"/>
    <xs:element name="Types" type="ws4d:QName-list" … />
    <xs:element name="DevRef" type="ws4d:EPR-list" … />
    <xs:element name="DevAdr" type="ws4d:QName-list" … />
    <xs:element name="CheckRef" type="ws4d:EPR-list" … />
  </xs:sequence>
</xs:complexType>
…
<message name="discoveryRequest">
  <part name="discoveryInput" element="ws4d:DiscReq" />
</message>
```

```
<variables>
  <variable name="svcDisc" element="ws4d:DiscReq" />
  <variable name="svcRef" element="ws4d:EPR-list" />
</variables>
…
<invoke partnerLink="discoveryServiceRequest"
  portType="ws4d:discSvcReqPT" operation="discoverService"
  inputVariable="svcDisc" outputVariable="svcRef" />
```

Fig. 6. Excerpt from the discovery service WSDL and its use in a WS-BPEL process.

D. WS-BPEL engine design considerations

The extensions to the WS-BPEL specification require also changes to the underlying WS-BPEL engine which executes the process. A discovery-enabled WS-BPEL engine must support the protocols WS-Discovery, WS-MetadataExchange and WS-Transfer which are used for the discovery of devices and services. Furthermore, it must comply to the discovery and validation procedures described above. The specified faults must be thrown if discovery does not return results or if the service validation evaluates to false.

Furthermore, the discovery-enabled WS-BPEL engine must check the declarations and usage of the three extension activities. It includes a validation against the XML Schema definition but also a *static analysis* of the process code as not all errors are detected by XML Schema validation. Beside the rules described in the subsection III-C an additional static rule is important. Web services being abstractly declared in a WSDL must not be invoked before they are discovered.

E. Concept for a discovery service

The discovery concepts can also be provided by a dedicated Web service. This is very useful for legacy WS-BPEL engines which might not support discovery Web service protocols nor presented discovery extensions. Instead of performing the discovery on a WS-BPEL engine by extension activities, the discovery and validation operations are invoked on the dedicated discovery service by the process.

Figure 6 shows an excerpt from the WSDL definition and the invocation of the operations from the process. The search criteria are wrapped into an XML Schema element and passed

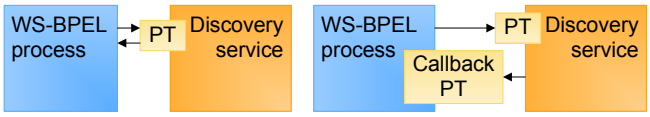

Fig. 7. Synchronous and asynchronous communication to the discovery service.

to the corresponding operation of the dedicated discovery service. The discovery service performs the discovery process as required and returns the results to the process.

The invocation of the discovery service can be performed in two ways as shown in figure 7: synchronous and asynchronous invocation. Just like using a discovery extension activity, a callback interface (`portType`) must be provided for the response to an asynchronous invocation which is received by a `<receive>` activity. The result of a synchronous invocation is returned in the variable `svcRef`.

The static analysis is indirectly performed by the discovery service. It takes care of handling any violations by rejecting the invocation request and returning an error message.

IV. Implementation of a Prototype

This work had to be implemented on a WS-BPEL engine which is compliant to the underlying Web services protocols. It appeared to be more difficult to adapt a WS-BPEL engine to the protocols than the protocols (DPWS toolkit) to the WS-BPEL code. The DPWS toolkit is based on Java and the SOAP engine Apache Axis2 and includes a code generator which generates the stub (client) and skeleton (service) for a Web service from its WSDL [7]. The developer only implements the application logic.

We have implemented a prototype for a WS-BPEL code generator which generates Java code (the application logic) from the process description and the WSDLs of involved Web services. The WS-BPEL code generator is based on XSL transformation. Figure 8 shows the process of design and code compilation.

An abstract location service was designed which searches for localisation services (in our case GPS, UWB indoor positioning and Bluetooth tracking) and returns the endpoint reference of the most exact location service being available. Although all functionality can be provided by the service discovery activity, all three of them have been used for evaluation purposes. This process was implemented using the WS-BPEL code generator. The dedicated discovery service was also implemented which can be used instead of the discovery extension activities.

V. Conclusion and Future Work

This paper presented a discovery extension to WS-BPEL in order to support dynamic device and service discovery as proposed by the Devices Profile for Web Services (DPWS). Discovery brings more flexibility and dynamic into WS-BPEL as involved Web services do not have to be known at design-time anymore and they can be exchanged at run-time. There are a lot of use cases starting from using discovery in the fault handler of a Web service invocation up to designing

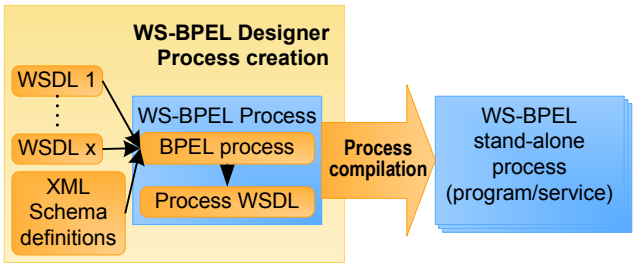

Fig. 8. Design and compilation of a WS-BPEL process.

flexible processes in industrial automation. We have shown the applicability of our presented approach and tested it in a prototypical implementation.

We are currently working on enhancing the discovery mechanisms with semantic approaches which will reveal the full potential of discovery in WS-BPEL. Also, development on a fully DPWS-compliant WS-BPEL is planned for the future.

References

[1] T. Erl, *Service Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, New Jersey, USA: Pearson Education Inc., 2005.

[2] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, *Web Services Architecture*, NOTE-ws-arch-20040211, W3C – World Wide Web Consortium, Boston, MA, USA, February 2004, URL http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[3] S. Chan, D. Conti, C. Kaler, T. Kuehnel, A. Regnier, B. Roe, D. Sather, J. Schlimmer (Editor), H. Sekine, J. Thelin (Editor), D. Walter, J. Weast, D. Whitehead, D. Wright, and Y. Yarmosh, *Devices Profile for Web Services*, Microsoft Corporation, Redmond, WA, USA, February 2006, URL http://specs.xmlsoap.org/ws/2006/02/devprof/devicesprofile.pdf.

[4] A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland, A. Guízar, N. Kartha, C. K. Liu, R. Khalaf, D. König, M. Marin, V. Mehta, S. Thatte, D. van der Rijn, P. Yendluri, and A. Yiu, *Web Services Business Process Execution Language Version 2.0*, OASIS Standard, OASIS – Organization for the Advancement of Structured Information Standards, Billerica, MA, USA, April 2007, URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[5] K. Ballinger, D. Ehnebuske, C. Ferris, M. Gudgin, C. K. Liu, M. Nottingham, and P. Yendluri, *Basic Profile 1.1*, BasicProfile-1.1-2006-04-10, WS-I – Web Services Interoperability Organization, April 2006, URL http://www.ws-i.org/Profiles/BasicProfile-1.1-2006-04-10.html.

[6] J. Beatty, G. Kakivaya, D. Kemp, T. Kuehnel, B. Lovering, B. Roe, C. S. John, J. Schlimmer (Editor), G. Simonnet, D. Walter, J. Weast, Y. Yarmosh, and P. Yendluri, *Web Services Dynamic Discovery (WS-Discovery)*, Microsoft Corporation, Redmond, WA, USA, April 2005, URL http://specs.xmlsoap.org/ws/2005/04/discovery/ws-discovery.pdf.

[7] E. Zeeb, A. Bobek, H. Bohn, S. Prüter, A. Pohl, H. Krumm, I. Lück, F. Golatowski, and D. Timmermann, "WS4D: SOA Toolkits making embedded systems ready for Web Services," in *Open Source Software and Product Lines Workshop (OSSPL07) in conjunction with the 3rd International Conference on Open Source Systems (OSS 2007)*, Limerick, Ireland, June 2007.

[8] T. Bellwood, S. Capell, L. Clement (Editor), J. Colgrave, M. J. Dovey, D. Feygin, A. Hately (Editor), R. Kochman, P. Macias, M. Novotny, M. Paolucci, C. von Riegen (Editor), T. Rogers (Editor), K. Sycara, P. Wenzel, and Z. Wu, *UDDI Version 3.0.2*, UDDI Spec Technical Committee Draft, Dated 20041019, OASIS – Organization for the Advancement of Structured Information Standards, Billerica, MA, USA, October 2004, URL http://uddi.org/pubs/uddi-v3.0.2-20041019.htm.

[9] D. Bachlechner, K. Siorpaes, D. Fensel, and I. Toma, "Web Service Discovery A Reality Check," DERI Digital Enterprise Research Institute, University of Innsbruck, Innsbruck, Austria, Tech. Rep., January 2006, URL http://www.deri.ie/fileadmin/documents/DERI-TR-2006-01-17.pdf.

[10] J. Garofalakis, Y. Panagis, E. Sakkopoulos, and A. Tsakalidis, "Contemporary Web Service Discovery Mechanisms," *Journal of Web Engineering*, September 2006.