Extending BPEL for Run Time Adaptability

Dimka Karastoyanova[3,1], Alejandro Houspanossian[2], Mariano Cilia[1,2],
Frank Leymann[3], Alejandro Buchmann[1]
[1]*Technical University Darmstadt, Darmstadt, Germany*
{cilia | buchmann}@informatik.tu-darmstadt.de
[2]*UNICEN, Faculty of Sciences, Campus Universitario Tandil, Argentina*
ahouspan@exa.unicen.edu.ar
[3]*University of Stuttgart, Stuttgart, Germany*
{leymann | karastoyanova}@informatik.uni-stuttgart.de

Abstract

The existing Web Service Flow (WS-flow) technologies enable both static and dynamic binding of participating Web services (WSs) on the process model level. Adaptability on per-instance basis is not sufficiently supported and therefore must be addressed to improve process flexibility upon changes in the environment. Ad-hoc process instance changes can be enabled by swapping participating WS instances, by modifying port Types of the partners to be invoked, and by changing process logic. In this work we address the problem of dynamic binding of WSs to WS-flow instances at run time, i.e. the ability to exchange a WS instance participating in a WS-flow instance with an alternative one. The problem is additionally complicated by the fact that the execution of a process depends on its deployment. We describe the "find and bind" mechanism, and we show its representation as a BPEL extension. We discuss the benefits that could be gained and the disadvantages it brings in. The mechanism extends and improves the existing process technologies. It facilitates a precisely controlled policy-based selection of WSs at run time and also provides for process instance repair, while maintaining simplicity. We also discuss a prototypical implementation of the presented functionality.

1. Introduction

Web Service Flows (WS-flows) are composite Web Services implemented using a process-based approach. Similarly to the traditional workflows WS-flows definitions specify declaratively collections of tasks executed by the participants in a process. A process definition also defines the execution order of tasks (control flow), the data exchanged among its tasks and its participants (data flow), exception handling, and business rules. Unlike the traditional workflows however, the WS-flows involve only a single type of participants – Web Services (WSs).

WSs are a technology aiming at the standardization of protocols and formats for the interaction among applications, in a language and platform independent manner, and even over the Web [38], [18]. WSs have *interface descriptions* in WSDL [37], which define the service functionality in terms of messages consumed and produced. WSs interact with each other in terms of *messages* and are *loosely coupled* [18]. This is a characteristic allowing the architecture to remain flexible to change [39]. In keeping with the SOA principles, WSs are *discoverable* [24], [5], [30], and can be *composed* in complex WSs. The technology has been designed with the purpose of application-to-application communication [18].

While WSs enable flexibility of organizations by defining standard protocols and formats, businesses often still use them in relatively simple scenarios. Boosting the technology acceptance and deployment depends partly on the ability to compose WSs in complex ones. Therefore the WS community, both from academia and industry, puts a lot of effort in specifying WS-flows. *BPEL* [9] is the *de facto* standard in this area but it does not support adaptability of WS-flows at run time. One approach towards *enhancing process adaptability* is the main topic of the paper, namely run time adaptations with respect to port binding in a per process instance manner.

WS-flows benefit from the features inherent to WSs [3], [17]. Having only WS participants makes WS-flow definitions independent from organizational structures and infrastructure specifics, i.e. regardless of formats and mechanism used to access the respective WS functionality. WS-flows provide a very flexible programming model relying on loose coupling to WS ports. Yet processes *flexibility* can be further enhanced: First,

Proceedings of the 2005 Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05)
0-7695-2441-9/05 $20.00 © 2005 IEEE

the ports (or WS instances) corresponding to the port types the process activities interact with are specified *upon deployment* in a static or declarative manner [19]; second, no run-time modifications can be done on the types of WSs that perform on behalf of the process or on the process structure. Therefore, in most of the cases in order to tailor a WS-flow to the constantly evolving environment one has to terminate the corresponding process instance, modify its model as necessary and restart it, even when the respective modification is needed only in a *single process instance*. Terminating and restarting long-running business processes is undesirable. The problem is aggravated by the fact that processes may take part in asynchronous communication with the participating WSs. Exception handling mechanisms are explicitly represented in BPEL [9] but they do not handle situations requiring dynamic swapping of WS instances (ports) at run time on process instance level. BPEL supports exchanging ports by allowing assignments of endpoint references from messages, but the assignment must be foreseen at the process model level. Replacing ports at run time *independently* of the process model and the predefined service selection policies is still not enabled. For this reason it is also impossible to repair a process (and its instances) in case the runtime environment is unable to find (and therefore enable interaction with) a service compliant with a portType prescribed by the process model. Related to this is the need for the ability to change selection policies even after process deployment; this is yet to be achieved for WS-flows, too.

In this paper we introduce an extension element to the BPEL language (version 1.1 [9]) that stands for a mechanism we call "find and bind". In general, this mechanism is meant to represent look up, selection and binding of services compliant with a port type defined in an activity in a WS-flow definition. This mechanism, even not at all new, has not been specified or represented by a language construct in BPEL. In the field of WS-flows it is only WSFL [20] that has a construct going for a similar approach. The "find and bind" mechanism is supported to some extent by most BPEL engines, but it is highly dependent on the engine implementation specifics and is tailored to reflect the engine-specific deployment of processes. Here we propose an approach toward making the mechanism explicit but in an optimal manner that, in addition, reuses and improves the existing practices. The proposed approach is not intended to substitute the existing process deployment procedures; it reuses them but also extends them to provide for deployment-independent process repair (see section 4). It is also a first attempt to generalize failure handling on instance level upon WS ports failure, as well as to enable selection policies modifications at run time. The mapping of the mechanism to a BPEL construct amplifies the adaptability of BPEL processes to changes in the environment on both domain-specific (laws, contracts, etc. changes lead to changes in the selection policies) and infrastructural levels (infrastructure failures, failing compliant WS instances).

The mechanism itself and its corresponding language construct are presented in section 3. The same section discusses the benefits and implications of using this mechanism and the language construct. Section 4 deals with the implementation of the mechanism in the context of the ReFFlow project [27]. The re-binding mechanism is mapped to a BPEL extension construct for adaptability. We use BPEL because of the acceptance rate of that language. The "find and bind" functionality is implemented as an extension to a BPEL engine, in this case the open-source ActiveBPEL engine [2]. Conclusions and future work are presented in section 5.

2. Adaptability and WS-flows – problem domain, state-of-the-art and motivation

WSs are used to provide organizations with *flexibility*. WSs render a heterogeneous environment homogeneous by hiding implementation specifics behind standard, stable WS interfaces. However, business organizations can be given greater flexibility and agility in supporting their business processes by letting them change not only the implementations of their simple WSs [39] but also by defining mechanisms they can use to adapt their complex/composed WSs, i.e. their business processes. Since WSs are the only type of performing entities in WS-flows, neither organizational models, nor infrastructure implementation specifics need to be considered in the WS-flow definitions. As described in [17] this simplifies to some extent the way WS-flows adapt to changes in the environment. Still, adaptability of WS-flows is not yet adequately supported.

Two major factors limit *adaptability* of WS-flows:

First, the existing WS-flow definition languages (e.g. BPEL [9], WSFL [20], XLANG [31]) assign potential participants, i.e. concrete ports, during process deployment to the process models in terms of port types. The portTypes are resolved into ports either upon deployment or during process

execution. A much more controllable and hence flexible approach is to enable the dynamic run-time look-up, selection and binding to WS instances on process instance basis. Instrumental here is the explicit and precise control over the selection of ports in terms of user defined criteria, all this at run time for each process instance. So far, this is only partly facilitated on the process level only (see details later in the section).

Second, changeability of WS-flows schema, in particular modifications of control and data flow, is not at all supported so far.

Research in the field of *adaptable workflows* has resulted in various classifications, mechanisms and approaches related to process adaptability [1], [7], [11], [12], [28]. [11] describes the layers of change together with taxonomy of approaches to workflow adaptation. Van der Aalst et al. [1] classifies possible modifications in the dimensions of a workflow on both schema and instance level in reaction to environment changes. Adaptability has never been considered as a part of the process meta-model, not that there has ever been a common, generally accepted meta-model for workflows [3], [6]. Instead, workflow products vendors have developed their own models and languages, that usually did not interoperate [6], [17]; adaptability has been enclosed in engine specific primitives.

WS-flows adaptability approaches are classified in [17]. Since WS-flow definitions are independent of organizational and infrastructural changes they need not adjust to changes in organizational models and/or implementation of participants' services; hence the reduced number of change-relevant layers in the classification (as compared to [11], [1]). Still WS-flows need to react to changes on the meta-model level, to changing conditions pertaining to a specific application domain, and to changes calling for control flow (and data flow) modifications [17], [15], [28], on both the process schema level and on instance level.

Meta-model and domain specific modifications in both workflow and WS-flow fields have been inadequately addressed; they are however outside of the scope of this paper. And while there is work done in enabling structural process schema changes in workflow products (time-based approaches - versioning, variants) that can be leveraged for modifying WS-flow models, the problem of ad-hoc changes in running *process instances* has not been dealt with.

Process instance modifications in an ad-hoc manner are related to the term *dynamic binding*. In principle, there are two types of entities that can be dynamically bound to a WS-flow: these are the types of participants (port types) and the actual participants (ports/WS instances compliant to a given port type) [17]. A possible preliminary approach to tackling the former case is presented in [15] and [17]. The state-of-the-art related to the latter case and the problems faced by the existing solutions, both language specifications and implementations, are discussed next.

The binding of WS-flows to ports at run time has been addressed in WSFL [20]. The language specifies a construct, called *locator*. The locator is nested in the `serviceProvider` element, which specifies the provider name and type. The locator determines whether a specific service is used for the execution of a task or a more dynamic approach is employed. For the latter case the locator enables two possibilities: a query to a UDDI registry or a mobility option. The uddi type locator also specifies the point in time of querying a UDDI registry [24] for compliant ports – startup (upon process instantiation), first hit (the first time an operation of the service provider is needed) or deployment (during the deployment of a process model) – and the criteria for selecting a service. In the mobility type binding the information needed for binding to a service provider is obtained as a result of a previous data exchange. Although still in use WSFL is officially substituted by BPEL.

BPEL [9] is the de facto standard for WS-flows. It supports a very flexible programming model (see [22] for an overview). A process definition models tasks (activities), control flow, and exception handling. Additionally, the process is exposed as a set of WSs and specifies all portTypes it implements. The specification does not deal with associating ports to port types. Typically, this association is a matter of deployment [21]. A process definition associates participants to a process on an abstract level only by means of partner link types. However, since process deployment is not standardized because of its dependence on the execution environment, the BPEL specification assumes that partner link types must be resolved upon deployment. An alternative to assigning participants to a process at deployment time is the other possibility BPEL includes – the so-called *mobility* notion (similar to the one in WSFL). In this case a process specifies that a partner in the interaction should send an endpoint reference during [36] the process execution. This endpoint reference (EPR) is to be used to interact with another participant. Again, this should be represented in the process model.

The BPEL specification allows all the information needed for dynamic binding of WS instances to be kept separate from the process definition. For this reason and because deployment

simply is not standardized the available BPEL engines employ different approaches.

The *current practices* [13], [14], [2], [19], [21], [26], [38] show that during *deployment* a port type gets assigned either a fixed port or a query. A query is used at runtime to determine a matching port according to predefined selection policies.

All BPEL implementations support *static* deployment [13], [2], [14], [26]. They all support the option of assigning ports to portTypes explicitly during deployment. The issue here is how to avoid terminating such process instances at run time, should the service bound to an activity fail.

Some BPEL engines support *dynamic* binding and allow assigning ports to portTypes of a process model at run time [2], [14], [26]; all of them use the approach based on deployment descriptors. Usually a query to a discovery component is performed (directly or delegated to the Service Bus [8]) and as a result a port (WS instance) compliant with the given portType and the predefined selection criteria is bound to the process. Deployment is assisted either by creating process deployment descriptors (e.g. WebSphere Business Integration Server Foundation Process Choreographer [14], Active BPEL [2]) or by special purpose tools or graphical interfaces that gather the required information (e.g. BPWS4j [13]).

Two potential problem issues must be considered here. First, it is possible that for some reason the execution environment signals a *failure to invoke the WS instance resolved as a result of the query and inability to find any other WS instance compliant with the portType and the selection criteria*. Process repair upon such a failure is generally not treated by the existing BPEL implementations and would lead to terminating the process. For such a case there is a need for a method to avoid terminating the process instance and assign a new portType-compliant port according to slightly different (and possibly relaxed) selection criteria. Second, whenever the *selection criteria* for a running WS-flow instance *change*, which is a plausible case, the information given during deployment with respect to the service selection policy become inadequate and would have to be changed. At present there is no straightforward way to bind a process instance to a participant, compliant to modified selection policy – hence the process instance must be terminated and then redeployed with the new policy data. Even if one changes the process's deployment descriptor, especially the selection policy there, this would not influence only the particular WS-flow instance, but rather all instances of the process model. This means that the set of ports used by an instance of a process model is fixed, in general. This again requires the ability to change ports during run time on per process instance basis.

As mentioned above, BPEL allows a process instance to receive a message from a partner that includes an endpoint reference to a WS. This enables the process instance to interact with a WS that has been unknown prior to the event of receiving that message. However, there is no prescription of what is to be done when the partner that is supposed to provide an endpoint reference fails. Whenever the WS sending the endpoint reference, to which an activity must subsequently be bound, is lost (failed, etc.) the process must be repaired manually by an administrator. This is actually another situation we would like to avoid.

All the above considerations call for a mechanism that allows for *dynamic binding* of ports to process instances at run time *independent of the policies provided upon deployment and without changing the process (model) itself*. Otherwise the process instances would have to be terminated for repair. This mechanism should reuse the extant specification and practices, and extend and improve them towards ensuring finer failure handling and thus greater adaptability. Additionally, such a mechanism should give the users the freedom to modify the selection criteria according to which the ports are selected even at run time.

In summary, even though the BPEL specification provides a very flexible basis for the development of WS-flows with dynamic features there is still ground for improvement. Therefore, in the next section we introduce the "find and bind" mechanism. Our motivation for defining and implementing this mechanism for WS-flows is the flexibility that can be gained when tackling the afore-mentioned issues and thus avoiding undesired process instance termination. The essence of the mechanism tailored to the WS-flows environment, the distinct problems it addresses, its representation in terms of a language construct, and the way it is implemented are the main contributions of this paper.

In the next section we briefly introduce the steps of the "find and bind" mechanism. We comment on the implications of using the mechanism and show its explicit representation as a language construct. We discuss the implementation in section 4.

3. Find and Bind – the mechanism and a language construct

This section describes the "find and bind" mechanism. The use of the mechanism is proposed in [17], [3] for providing the users with the

flexibility to choose explicitly at run time the WS instances participating in a process instance.

The idea is not a new one: dynamic look up of components and dynamic binding has already been applied in the middleware technologies [3], and has been described in WSFL [20] for WS-flows. But corresponding support in BPEL is missing. All ports participating in a WS-flow are resolved upon deployment or at run time based on predefined selection policies assigned to the process model. WSs instances locations are either hard-coded in the process WSDL description or in a deployment descriptor, or determined as a result of a query to a registry (see also section 2). The run-time search and selection of WS instances in a deployment-independent manner for running process instances is not yet a part of any of the languages for WS-flows, nor is it supported by any of the existing BPEL engines. In other words, current specifications do not address the issue of ad-hoc process changes on per-instance basis adequately enough. In order to enable selection of ports at run time according to selection criteria different from the ones specified by the deployment descriptor we introduce an additional element to the BPEL language.

3.1. Steps to perform – find, select and bind

In general, the "find and bind" mechanism comprises three main steps:

1. Find a list of all available WSs compliant with the portType specified in an activity of a process;
2. Select a port from that list according to user-defined selection criteria (QoS, semantics);
3. Bind the activity of the process instance to the selected port. This port is the one that is going to perform a task on behalf of the process instance under consideration.

The general form of the mechanism is presented in the next figure.

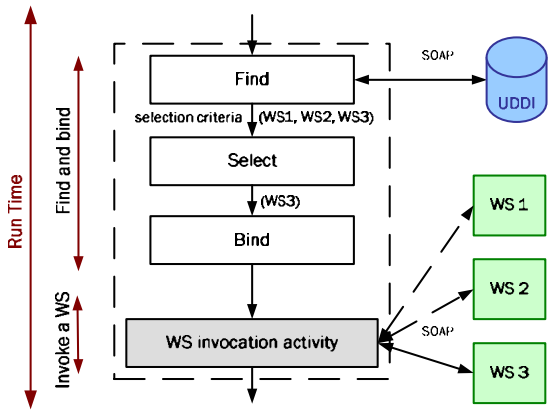

Figure 1. "Find and bind" mechanism

As pointed out in [17], [15] there are reasons to make the mechanism explicit in a unified WS-flow meta-model and expose it to the WS-flow users (administrators, developers) in terms of a language construct. We choose to represent it in terms of the `<find_bind>` construct in a common process definition language. Because of the current trend towards using BPEL [9] for WS-flow definitions it is only reasonable to extend the BPEL language with the `<find_bind>` construct. The process developers and users do not have to care about how the "find and bind" mechanism is performed and therefore this must be reflected in the model elements and the corresponding language construct; they only should care and know about the selection criteria and specify them within the `<find_bind>` construct. This construct is designed to express declaratively the requirements towards a WS port in terms of selection policies. Users (administrators) are thus given the ability to specify *default values* of selection criteria that are to be used at run time as a substitute of those given during process deployment or as their extension and refinement. This is especially useful in the case of process instance repair.

In BPEL the activities standing for interaction with Web services are the `<invoke>`, `<receive>`, and `<reply>` activities, hence they are the ones to be extended with the `<find_bind>` extension element. The next code listing shows an example of a `<find_bind>` element for an `<invoke>` activity; note that this additional element is included into the set of standard elements of the activity.

The example in Listing 1 shows that the "find and bind" mechanism is mapped to a separate extension element of an interaction activity. The element includes a "`selection_policy`"

attribute. During the process build time one specifies default values for the selection policy (in order to control the selection of compliant services in the case of a failure).

```
<process name="ConvertCurrencyBP">
<!-- details … -->
<invoke
  name="ConversionRequest"
  partnerLink="converter"
  portType="CurrencyConvService"
  operation="usd2eur"
  inputVariable="C_and_Rate"
  outputVariable="result">
  <find_bind
      selection_policy =
           "selection_policy_ID"/>
</invoke>
<!—details … -->
</process>
```

Listing 1. Example of the <find_bind> extension element

The value of the `selection_policy` attribute can either directly specify the selection policy in terms of a list of criteria or it can be a name of a selection policy. The name uniquely identifies a selection policy and can be used to reference policies stored in a separate infrastructure component. Therefore we distinguish between in-line policies and referenced policies.

Using the "find and bind" functionality during process instance execution every time a WS has to be invoked is not the most optimal solution even though it might bring greater flexibility due to the fact that the set of ports compliant with the policy changes all the time even during the execution of a single process instance. Moreover, there are much more optimal solutions with respect to performance which handle most of the cases for process configurability (see section 2), namely all points of variability are specified separately upon process deployment. As we mentioned above, during deployment of BPEL processes activities get endpoint references of participants or selection policies associated.

Nevertheless, the following cases are not being covered by the deployment approach; we discuss next how they could be addressed using the `<find_bind>` extension construct.

3.1.1. Process instance repair. In the case of a *failing port and inability of the runtime environment to find any other port compliant* with both the port type and the selection criteria defined upon deployment the process instance must be interrupted and repaired. Usually the repair on process level is not an automated procedure and is done manually by administrators. If the undertaken repair action involves modification of a deployment descriptor (the deployment information in general) it will inevitably influence all instances of the repaired process. If, on the other hand, the `<find_bind>` element had been included in the activity that needed to be repaired, with default selection policies, which are alternative to the criteria specified in the deployment descriptor, the engine would be able to perform another query and select another compliant port. Thus terminating the process instance that has experienced such a system failure is avoided, and no other instance is influenced by the repair. The additional benefit here is that the repair is automated. In our view, this is a way to automatically bypass the predefined selection criteria (and in a way to overwrite endpoint references) for a process instance only and whenever needed. In essence this approach enables *deployment-independent service selection at run time*.

An additional point we would like to make here is that the port used by the process as a result of the repair could also fail. In this case "find and bind" must be performed again. It should be repeatedly executed until the interaction succeeds (i.e. the activity is executed) or as long as there are ports in the list of compliant ones. After that a manual repair is required. Manual repair would also be required if for some reason the execution of the "find and bind" functionality fails – e.g. the discovery component is not reachable, there are no other ports implementing the originally specified port type, etc.

3.1.2. Modifying selection policies. There is another problem calling for the use and explicit representation of the "find and bind" mechanism not covered by the BPEL specification and any of the extant implementations. The solution improves process adaptability to an even greater extent. Since selection criteria are specified upon deployment it is not yet possible to adjust a process instance in accordance to changing user requirement towards the selection of services; any changes in the policies affect all instances. Policies for choosing WS instances are also an *artifact of change*. By representing selection policies explicitly in the `<find_bind>` construct and with the necessary tools available one could also *change* the default *selection policies*. Thus users would be given an extra ability to guide the selection of ports at run time depending on their needs and according to the changing rules of their business. This is possible by using just monitoring tools that support viewing of process instances and allow for changing the selection policy at run time.

We propose making the mechanism explicit only for the purpose of enhancing and improving the language expressiveness and extending the existing practices. We by no means propose substituting the existing approaches; our intent is to improve the existing state-of-the-art but in the same time draw on the existing approaches and experience. The explicit language element allows for deployment-independent service selection at run time on a per instance basis and is a first approach towards standardized handling of exceptions of that kind.

We also envision the application of this extended mechanism for dynamic selection and binding of WS instances in advanced WS-flow engines. In the context of the ReFFlow methodology, which considers the development of an advanced features BPEL engine [15], [27], no references to ports and even no portTypes need to be specified in the process definition nor are they assigned to the process prior to run time. In that case the functionality of the `<find_bind>` construct (i.e. the mechanism it stands for) must be supported by the engine and performed at run time for each activity that invokes a WS for which the portType and operation values remain unknown until just before their execution.

3.2. Implications, advantages and disadvantages

Using the `<find_bind>` construct to make WS-flows more flexible has implications as concerns tool support, performance cost, handling faults, description of WSs semantic and quality and so on.

It is of benefit that *flexibility* is gained while *simplicity* is preserved. The `<find_bind>` element contains no implementation-specific features – no reference to an implementation language, platform or discovery component to be used for the look up (compare to WSFL locator, where only the UDDI registry was considered an option). These features remain hidden because of the declarative nature of the definition language, preserved in the `<find_bind>` element as well. It is the process engine that has to tackle the actual search for compliant ports and has to resolve the selection criteria. Typically, the process engine will delegate this to another infrastructure element, and eventually bind the process to the selected instance. The so-called Enterprise Service Bus [8] could implement such an infrastructure component.

As we stated earlier, the user gains additionally the benefit of being able to guide the selection of the participating WSs and thus adapt their processes by *modifying the criteria for the selection* even at run time. In the context of autonomic computing the modification of selection policies could be made automatic as well. However, while users are allowed to prescribe default selection policies, they depend on the *availability of* appropriate *tools* for modifying the selection criteria at run time. Such tools should allow users/administrators to actively control the selection of WS instances. Having in mind the current trend toward using BPEL for WS-flows these tools should support editing of BPEL models and/or files. These tools could partly enforce the correctness of modifications by permitting only a predefined set of valid changes. An additional tooling feature would be to support process deployment.

The "find" step of the mechanism requires calls to a discovery component to get the list of WS instances conforming to the portType specified in an invoking activity. One such discovery component is the UDDI registry, but in general, any other discovery component or approach supported in a service bus environment will do [21], [3], [5], [30]. The support infrastructure must be able to interact with the discovery component and perform the necessary mapping to its internal format though.

Dynamic binding to WS instances faces an additional difficulty when the interaction between a process and a WS instance is asynchronous. Detecting a failure of a WS instance in the case of *asynchronous communication* means that the previous actions must be compensated for and (status) data resent to the newly selected WS instance. Handling failures and compensating for work done in an asynchronous interaction mode between processes and participants is an issue of critical importance and a part of our future research.

After a set of compliant WSs is found a single service instance must be selected. The selection criteria are either the default ones prescribed by the workflow developer within the `<find_bind>` element or new ones updated by the users while the process instance is running. The infrastructure component that is responsible for the selection gets as input the identifier of the selection policy as stated inside `<find_bind>` or a list of criteria, and their values, and the list of services compliant with a portType. Its task is to find those services from the set whose policy descriptions match the selection policy. While the `<find_bind>` element defines which selection policy is to be used, it does not reveal any details about the infrastructure entity that performs the actual selection. Therefore the *selection functionality* can be assigned either to the engine, or implemented by a separate stand-alone component that can be

(re)used by various WS-flows engines; it might also be available as a WS. Obviously, an additional task of the selection component is finding out what the policies of the WSs in the list are. The output the component must produce is the choice it has made, in particular the endpoint reference of the WS instance it has picked out. Further details on the selection component are outside the scope of our current discussion. However, it is important to note that the development of the selection module depends on the availability of standards for semantic description of WSs and for quality of service (QoS) models for WSs. Even though there are standards for semantic description of Web resources (RDF [35], OWL [34]) and WSs (DAML [10] and OWL-S [23]) we are not aware of software products available that could be used to directly support the selection step of the "find and bind" mechanism. There is no agreed upon QoS model for WSs yet, but once it becomes available it can be used within the "find and bind" mechanism immediately. In any case, our point here is that the infrastructure must incorporate a module to handle *policy match-making and WS selection*.

Apart from the cases of binding repair at run time the "find and bind" functionality finds additional use together with the `<evaluate>` extension element, introduced in [17] and [27]. The `<evaluate>` element is used to ensure that processes defined without any reference to portTypes in activities will be deployable and executable. It is also meant to support modification of statically provided portTypes in BPEL processes at run time [17]. Because in both of the above cases the actual portTypes to be used during the WS-flow execution are unknown prior to the execution of the corresponding activities themselves the dynamic search, selection and binding to a compliant WS port is an obvious prerequisite.

The "find and bind" mechanism is mapped on an element that is a part of the standard elements section within the `<invoke>`, `<receive>` and `<reply>` activities, rather than on a separate activity type. This choice is substantiated by several facts. Most process *modeling tools* are graph-based. Introducing a new activity type would mean a incorporating the representation of that activity type in all modeling tools that import and export BPEL files. This definitely hampers portability of BPEL files across multiple modeling and editing tools. Another issue here is *performance*. The navigator module of a workflow management system would have to navigate through an additional activity should the "find and bind" mechanism have been mapped on an activity type. This would have led to performance penalties in addition to those already imposed by the calls to a discovery component, in particular if it is a UDDI registry (because of its characteristics).

4. Functionality implementation

In this part of the paper we present how the "find and bind" mechanism has been implemented in the context of the ReFFlow project [27].

There are several BPEL-compliant engines, some of which are open source ones, e.g. Twister [32] and ActiveBPEL [2]. While both Twister and ActiveBPEL provide support for all major BPEL features, we have chosen to extend the ActiveBPEL engine with the "find and bind" functionality because of the available documentation, its development status, and its large number of auxiliary features.

The `<find_bind>` construct does not incorporate any implementation specific features and therefore using and enhancing any other BPEL compliant engine would have also been possible. It is important to extend the engine without modifying its original code. We use the aspect-oriented programming (AOP) approach for this purpose. This approach allows for modular and adaptable extensions, which can be weaved in the execution flow any time they are needed, while the original source code from ActiveBPEL.org [2] is preserved; besides, this allows us to use any new releases of the engine and still be able to extend them accordingly.

The major components of the Active BPEL architecture are in charge of process representation (at deployment and run time) and process execution, event and alarm handling, and WS invocation management. The engine runs on various application servers and relies on AXIS [4] for dealing with SOAP messages. To facilitate the discussion further, in the next sub-section we briefly introduce how processes are represented and executed in ActiveBPEL.

4.1. Engine characteristics

Before we present the implementation of the extended functionality, first we pay attention to those characteristics of the Active BPEL engine relevant to it. We also discuss briefly what auxiliary modifications had to be done to support this functionality. Figure 2 depicts how the ActiveBPEL engine interprets and instantiates BPEL processes. Three steps are performed:

(A) parsing the XML file and creating a DOM representation,

(B) generating internal representation of the process definition,
(C) creating business process instances.

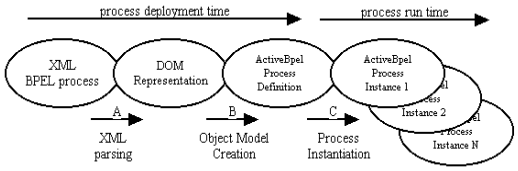

Figure 2. Process representation in the ActiveBPEL engine

Having the above procedure in mind additional functionality must be appended to support parsing the `<find_bind>` element correctly and generating the internal process definition with the corresponding "find and bind" functionality. Since the extension element has a DOM representation unknown to the available ActiveBPEL engine the reader module that interprets the DOM representation of the `<invoke>`, `<receive>` and `<reply>` activities has been modified to account for it in the internal process definition. This is implemented in terms of aspects executed only when a `<find_bind>` has been encountered in a definition. The actual data extraction is trivial. The default selection policy information must also be extracted and stored in a separate data structure.

In ActiveBPEL there is an invoke handler that executes an AXIS call [4] on behalf of activities interacting with WSs from within a process instance. The information needed to perform the invocation (partnerLink, portType, operation, port location, etc.) is provided by the attribute values of the invoking activity. If an Axis fault is detected by the handler, it notifies the activity of the fault. The activity notifies its process of the fault and the process, in turn, notifies the engine. Typically, when a WS invocation fails, the engine stops the process and reports the error to the client of the process. To avoid this and allow for the process to complete when a WS has failed, the current engine functionality needs to be extended to perform additional search for compliant WSs and bind to one of them. Therefore the main extension work is done on the invoke handler. Its functionality is extended in such a way that it first performs the "find and bind" mechanism (i.e. it finds a compliant port and its location) and then invokes the service by performing an AXIS call to the dynamically selected service instance.

In brief, upon detection of a WS failure at run time the "find and bind" will be executed. A fault in a WS invocation can be tackled in terms of the "find and bind" mechanism if and only if an invocation activity contains a `<find_bind>` element as specified above. Otherwise the default fault handling applies.

4.2. Find & Bind implemented

A detailed description on the implementation of the "find and bind" steps is given next. As it was mentioned above, the find and bind functionality is decomposed as follows (see also Figure 1):

- Find: Given a port Type name a list of published compliant WS instances is returned.
- Select: According to the given selection criteria a single WS instance is selected from the list of services returned by the "find" step.
- Bind: The endpoint reference of the original WS instance is replaced by the new one and a call to the WS is executed.

4.2.1. Find. During this phase we aim at finding a list of services that implement a particular port Type. The discovery component we employ so far is a standard UDDI [24] registry. It is queried using UDDI4j [33] – a Java implementation of the UDDI API. A mapping between the WSDL and UDDI data models is needed (as specified in [25]) because the UDDI Inquiry API does not work in terms of WSDL elements (such as the portType name) but rather in terms of the UDDI entities.

First, a UDDI4j proxy against a UDDI registry is created. The SAP [29] UDDI registry is used here. The class UDDIProxy provides find_XX methods, whose execution implies the execution of a query against a remote UDDI registry [33].

```
UDDIProxy proxy = new UDDIProxy();
proxy.setInquiryURL("https://uddi.sap.
com/UDDI/api/inquiry/");
```

Second, we have to get the tModel key of the portType specified by the invoking activity:

- Look for the tModel that has the appropriate portType name and is categorized as "portType" according to the WSDL ENTITY TYPE category system [25]. The categorization is expressed using a UDDI CategoryBag, populated appropriately with UDDI references.
- Retrieve the tModel's categorization key.

```
CategoryBag categorization =
                    new CategoryBag();
categorization.add(new KeyedReference
("WSDL Type", "portType", WSDL_ENTITY_
TYPE_category_system_tModelkey));
TModelList tModelList =
proxy.find_tModel(portTypeName,
categorization, null, null, max_rows);
TModelKey ptTModelKey =
tModelList.getTModelInfos().get(0).get
TModelkey();
```

- If the portType has a target namespace it must be referenced as part of the categorization, i.e. the category bag must be populated with an additional key referencing the namespace that defines the portType [25].

The third task is to look for the Services that implement the portType:

- Find all services with the portType key determined in the previous step. The list of services that match this criterion can be retrieved using the find_service function.

```
TModelBag tmb= new TModelBag();
tmb.add(ptTModelKey);
ServiceList serviceList =
proxy.find_service(null, null ,null,
tmb, null, max_rows);
```

Fourth, retrieve the complete information about the services in the list.

- Create an auxiliary data structure "serviceKeys" containing all service keys from the list.
- Get details for each service, which is actually the data needed for the selection step:

```
businessServices =
(proxy.get_serviceDetail(servicesKeys)
).getBusinessServiceVector();
```

The first two steps could be accomplished in one single step. The find_service_XX function of the UDDI API provides the chance of executing an embedded find_tModel call. In this way, we would have directly executed step 2, with step 1 as embedded query. This would have reduced the number of calls to the UDDI registry. But UDDI4j (v2.0.2) does not support embedded queries as part of the find_service_XX function.

4.2.2. Select. During the selection step a WS instance is chosen out of a set of WS instances compliant with a portType. The selection is guided by the (default) selection policy. Policies are derived from the data provided as part of the <find_bind> element. A selection policy can be specified in terms of a policy identifier or a set of criteria. The policy can describe semantic characteristics of a WS and/or QoS features. The semantic and QoS descriptions of WSs are a hot research topic together with matchmaking of policies for WSs. However, the available languages and techniques are either not completely specified or not agreed upon, and there are no supporting tools that one could use directly as a part of this engine extension. However, the <find_bind> element is designed and implemented to accommodate future advances in these research areas.

Currently the selection module is implemented to select a single WS instance based on a simple policy that just picks randomly a service from the list returned by the "find" module. In our future work we intend to extend the selection module to define and apply more complex policies based on QoS and/or semantic descriptions of the WSs. This as we said before is dependent on the standardization initiatives related to WSs semantic descriptions and QoS models for WSs.

The selector module instantiates a policy object using the policy information provided by the <find_bind> element. It performs the WS instance selection using the policy and returns the binding information for the selected WS:

```
ISelectionPolicy policy =
policyManager.instantiatePolicy(select
ionPolicyInfo);
BindingTemplate bindingInfo =
policy.selectWS(services);
return (bindingInfo);
```

4.2.3. Bind. The binding mechanism is engine dependent. Essentially, the "bind" step implies replacing the endpoint reference (in our prototype, this is basically a URL) of the service that has failed with a new one so that the call to the alternative service instance is made possible.
The bind functionality in the extension gets as input the binding information for the selected WS instance.

The URL is extracted in the following way:

```
URL newURL = new
URL(bindingInfo.getAccessPoint().
getText());
```

The returned URL is assigned to the AXIS call object the engine employs for the invocation:

```
call.setTargetEndpointAddress(newURL);
```

4.3. Experiments

The test set-up for the implementation utilizes a simple WS-flow that implements a currency converter. It invokes two different WSs: one for getting the cross-currency rate, and another for calculating the conversion given the amount and rate. In the definition both invoking activities included a <find_bind> element. We deployed the process on the enhanced engine with all specified participating WSs available. The process executed without involving the extended functionality because even though the <find_bind> construct was present in the definition no WS failed; hence no re-binding was necessary. In the next experiment we deployed the process and made the WSs unavailable; thus we created a condition that would trigger the process

repair with "find and bind". Here, an AXIS fault was detected, which triggered the "find and bind" functionality. During the find phase a UDDI registry was queried, alternative WSs URLs were selected and bound to the invoke activities. The process instance was able to complete its execution without reporting a fault that would require its termination.

5. Conclusions

The existing practices enable dynamic binding of ports to processes by specifying criteria for port selection during process deployment. However, since deployment is execution environment specific it cannot be standardized in a cross-platform manner. Moreover, selection policies given during process deployment are valid for all instances of the process model and can be modified only when processes are redeployed. Because of these limitations, a mechanism that supports deployment-independent ports selection at run time on process instance level is needed.

In this paper we described the "find and bind" mechanism as an approach to enabling dynamic binding of WSs to WS-flow instances at run time. It is meant to enhance and improve the existing techniques for dynamic binding of ports to WS-flows. It draws on their optimized performance and practical significance and improves them further. It enables process instance repair without interrupting the process instance upon port failures. Presenting the selection policies explicitly in the process definition is a premise for adjustment of processes to changing user requirements towards the WSs participating in a WS-flow.

Again, it is of utmost importance to give the users the control over the selection of WS instances during process execution on per instance basis using precise selection criteria. At the same time the implementation specifics of the execution environment remain transparent, by employing the declarative nature of the WS-flow languages.

The mechanism has been implemented as a part of the ReFFlow infrastructure [27]. It has been represented as an extension of the BPEL language. In this paper we also presented the implementation of the mechanism. Additionally, the implications of using the mechanism itself and its language representation, both positive and negative, were extensively discussed.

The binding of WS-flow instances to WSs in an ad-hoc manner is only one of the approaches towards achieving WS-flow flexibility. In the ReFFlow project another meta-model extension construct is being developed. It is the `<evaluate>` extension element [17] and it enables modification of port types at run time; as a future work reference it is also meant to facilitate changes in process control flow. This construct makes extensive use of the concept of templates [16] to boost reusability and to help toward greater flexibility and faster process adaptation.

Our future research activities include also work on accommodating more complex selection policies and the development of an infrastructure component able to handle WS instance selection based on such sophisticated criteria.

The dynamic binding of WSs is not a problem pertinent to only WS-flows but rather to the WS technology as a whole. It is one approach to enabling loose coupling and its numerous advantages in a SOA [8], [18]. Future trends in this respect include the creation of a sophisticated infrastructure to support advanced dynamic features of service-oriented environment – this is the so-called Service Bus.

6. References

[1] van der Aalst, W.M.P., Jablonski, S.: Dealing with workflow change: identification of issues and solutions. *International Journal of Computer Systems Science and Engineering*, Vol. 15, No. 5, September 2000. CRL Publishing Ltd.

[2] ActiveBPEL Engine. http://www.activebpel.org/

[3] Alonso, G., Casati, F., Kuno, H., Machiraju, V.: *Web Services. Concepts, Architectures and Applications*, Springer-Verlag. Berlin Heidelberg New York, 2003.

[4] Apache Axis: http://ws.apache.org/axis/

[5] Ballinger et al.: Web Service Inspection Language (WS-Inspection, WSIL), November 2001. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-inspection.asp

[6] Baeyens, T.: The State of Workflow. TheServerSide.com, May 2004. http://www.theserverside.com/articles/content/Workflow/article.html

[7] Casati, F., et al.: Adaptive and Dynamic Service Composition in eFlow. Proceedings of CAiSE 2000, LNCS 1789, Springer Verlag, 2000.

[8] Chappell, D.: Enterprise Service Bus. June 2004, O'Reilly

[9] Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., Weerawarana, S.: Business Process Execution Language for Web Services (BPEL4WS) 1.1. May 2003. http://www.ibm.com/developerworks/library/ws-bpel

[10] DAML.org: DAML Services. http://www.daml.org/services/owl-s/

[11] Han, Y., Sheth, A., Bussler, Chr.: A Taxonomy of Adaptive Workflow Management. In Proceedings of the "Towards Adaptive Workflow Systems" Workshop at the 1998 ACM Conference on Computer-Supported Cooperative Work (CSCW98), Seattle, 1998.

[12] Heinl, P., Horn, S., Jablonski, S., Neeb, J., Stein, K., Teschke, M.: A Comprehensive Approach to Flexibility in Workflow Management Systems. In Proceedings of WACC'99, 1999.
[13] IBM AlphaWorks, "IBM Business Process Execution Language for Web Services JavaTM Run Time (BPWS4j)", IBM, 2002, http://www.alphaworks.ibm.com/tech/bpws4j
[14] IBM developerWorks: WebSphere Business Integration Server Foundation Process Choreographer. http://www-106.ibm.com/developerworks/websphere/zones/was/wpc.html
[15] Karastoyanova, D., Buchmann, A.: ReFFlow: A Model and Generic Approach to Flexibility of Web Service Compositions. In Proceedings of iiWAS 2004, September 2004.
[16] Karastoyanova, D., Buchmann, A.: Automating the development of Web Service compositions using templates. In Proceedings of "Geschäftsprozessorientierte Architekturen" Workshop, at Informatik2004, Ulm 2004.
[17] Karastoyanova, D., Buchmann, A.: Extending Web Service Flow Models to Provide for Adaptability. In Proceedings of OOPSLA '04 Workshop on "Best Practices and Methodologies in Service-oriented Architectures: Paving the Way to Web-services Success", Vancouver, Canada, 2004.
[18] Kaye, D.: Loosely Coupled: The Missing Piece of Web Services. RDS Press 2003
[19] Kloppmann, M., König, D., Leymann, F., Pfau, G., Roller, D.: Business process choreography in WebSphere: Combining the power of BPEL and J2EE. IBM Systems Journal 43(2) (2004).
[20] Leymann, F.: Web Services Flow Language WSFL. IBM Corporation, 2001. http://www.ibm.com/software/solutions/webservices/resources.html.
[21] Leymann, F.: The Influence of Web Services on Software: Potentials and Tasks. Proc. 34th Annual Meeting of the German Computer Society (Ulm, Germany, September 20 – 24, 2004), Springer, 2004.
[22] Leymann, F., Roller, D.: Modeling Business Processes with BPEL4WS. Information Systems and e-Business Management (ISeB), Springer, 2005.
[23] Martin, D. et al.: OWL-S 1.1. November 2004. http://www.daml.org/services/owl-s/1.1/
[24] OASIS UDDI Specifications TC: UDDI v. 3 Specification. October 2003. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm
[25] OASIS Technical Note "Using WSDL in UDDI Registries v2.0.2"
[26] Oracle Corporation: Oracle BPEL Process Manager 2.0. 2004. www.oracle.com/technology/products/ias/bpel/index.html
[27] ReFFlow Project. www.dvs1.informatik.tu-darmstadt.de/research/refflow/index.html
[28] Reichert, M., Dadam, P.: ADEPTflex - Supporting Dynamic Changes of Workflows Without Losing Control. Journal of Intelligent Information Systems 10(2) (1998).
[29] SAP registry. https://uddi.sap.com/UDDI/api/inquiry/
[30] Skonnard, A.: XML Files: Publishing and Discovering Web Services with DISCO and UDDI. MSDN Magazine: The Microsoft Journal for Developers. February 2002. http://msdn.microsoft.com/msdnmag/issues/02/02/xml/
[31] Thatte, S.: XLANG: Web Services for Business Process Design. Microsoft Corporation, 2001. http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm
[32] Twister, 2004 www.smartcomps.org/twister/
[33] UDDI for Java: http://www.uddi4j.org
[34] W3C: OWL Web Ontology Language. Reference. W3C Recommendation, February 2004. http://www.w3.org/TR/owl-ref/
[35] W3C: Resource Description Framework (RDF). February 2004. http://www.w3.org/RDF/
[36] W3C: Web Service Addressing (WS-Addressing) Member Submission. August 2004. http://www.w3.org/Submission/ws-addressing/
[37] W3C: Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, W3C Working Draft, 2003. http://www.w3.org/TR/wsdl20
[38] Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.: Web Services Platform Architecture. Prentice Hall 2005.
[39] Westbridge Technology. Critical Infrastructure for Service-Oriented Architectures (SOA). 2004