Flexible Binding for Reusable Composition of Web Services

Cesare Pautasso, Gustavo Alonso

Department of Computer Science
Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, 8092 Zürich, Switzerland
{pautasso,alonso}@inf.ethz.ch

Abstract. In addition to publishing composite services as reusable services, compositions can also be reused by applying them to orchestrate different component services. To do so, it is important to describe compositions using flexible bindings, which define only the minimal constraints on the syntax and semantics of the services to be composed. This way, the choice of which service to invoke can be delayed to later stages in the life cycle of the composition. In the context of Web service composition, we refine the concept of binding beyond the basic distinction of static and dynamic binding. Bindings can be evaluated during the design, the compilation, the deployment, the beginning of the execution of a composition, or just before the actual service invocation takes place. Considering the current limited support of dynamic binding in the BPEL service composition language, we show how we addressed the problem in JOpera, where modeling bindings does not require a specific language construct as it can be considered a special application of reflection.

1 Introduction

Software components are – by definition – reusable [28]. In this paper we look at reusability from the opposite perspective and ask the following question: are compositions also reusable? Clearly, thanks to the recursive nature of most composition languages, compositions as a whole can be immediately reused as components. Web services, a particular kind of component used to build service oriented architectures [26], are also intended to be reused in many different combinations (e.g., [34]). Likewise, composite services are typically published themselves as Web services [24].

In addition to reuse based on packaging entire compositions as components, compositions – as opposed to basic components – can also be reused along a qualitatively different dimension. As it was informally exemplified by W. Tracz in [28]:

Part of this work is supported by grants from the *Hasler Foundation* (DISC Project No. 1820) and the *Swiss Federal Office for Education and Science* (ADAPT, BBW Project No. 02.0254 / EU IST-2001-37126).

> If you have components to reuse then you need to glue them together. If you have patterns to reuse, then you have the glue into which you have to stick pieces. After you glue pieces together long enough and you start seeing a pattern, then you can reuse the glue too.

Along these lines, the main contribution of this paper consists of applying such idea to Web service composition. In particular, we discuss what are the requirements for service composition languages, techniques and tools to support reusable composition. To this end, the notion of *binding* is very important, as it captures the relationship between the composition and its component services and defines what are the corresponding reusability constraints.

In practice, reusability is not the only issue related to the flexible binding of services into compositions. Other interesting ones concern the testability and the reliability of the compositions. These are also two very important aspects to be taken into account when building distributed applications out of the composition of Web services.

From a different perspective, current Web service technologies can be seen as an evolutionary step of existing RPC [4] based middleware to cover distribution at a World-wide-web scale [1]. In this sense, SOAP [29] is used as a wire-protocol, WSDL [30] as the interface description language (IDL), while UDDI [19] was meant to provide the foundation for a global registry infrastructure. Based on this, the focus of this paper is to discuss how to apply the notion of binding to address the Web service composition problem and to determine how well the existing Web services composition languages and tools support it.

Given the current limited level of support for static and dynamic binding in BPEL4WS [16], in this paper we present how reusable service compositions can be built using JOpera. JOpera is an open research platform for service composition developed at the Swiss Federal Institute of Technology. It features a visual composition language [22], a set of rapid composition tools based on Eclipse [20] and supports an extensible set of composition techniques [23].

This paper is organized as follows. We introduce the problem of binding in service oriented architectures by showing some of the limitations of BPEL in Section 2. Then, we refine the notion of flexible binding according to different orthogonal aspects: its scope and its evaluation time. In Section 4 we present how the JOpera system uses reflection to support flexible binding. In Section 5 we discuss related work in the context of Web service composition. In Section 6 we draw some conclusions.

2 Motivation

In service oriented architectures, binding is an abstraction mechanism to separate implementation specific details from a high-level description of the functionality of the services to be accessed. As an example, consider the approach followed by WSDL, which separates the abstract description of a service interface (the *port type*) from the transport protocols and the addressing information used to access

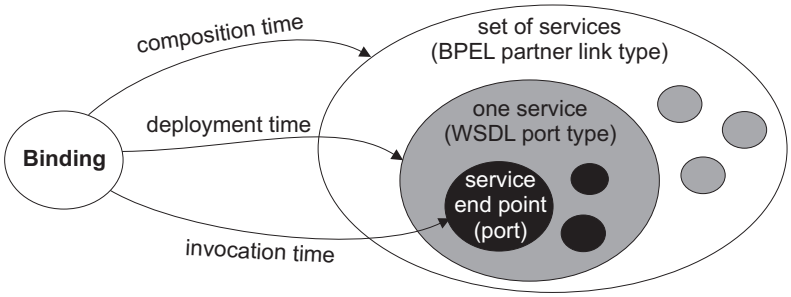

Fig. 1. Progressive refinement of a binding using BPEL

the corresponding service provider (the *port*). This separation of interface from implementation fulfills the important requirement of adaptability that make a composition reusable in an ever-changing distributed environment.

Building on this, BPEL[1] uses the notion of *abstract process* and *partner* to reflect the idea that the business processes modeling how different Web services should be composed can be applied to different service providers. By extending the WSDL standard service interface description with the declaration of a *partner link type*, the process which defines how to compose different services does not depend on specific services but can be customized to use different ones as long as they fit with the expected link type.

As represented in Figure 1, during the life time of a process, a binding is progressively refined going from a set of services to one specific service end point. When a process is designed, it does not contain any reference to a specific service, but it only lists partner port types. An abstract process is reused by constructing a mapping between concrete service port type definitions and the *roles* played by the various partners in the process. This mapping between WSDL definitions and partners is set once the composition is deployed and it is fixed for all executions of the composition.

As shown in the example (Figure 2), the only form of dynamic binding supported by BPEL consists of re-assigning end points which identify specific ports within a service interface at runtime. End points are identified with the WS-Addressing proposed standard [32]. Although this feature can be used to establish call-back relationships and allows to tap the flexibility offered by the separation between port types (interfaces) and ports (communication end-points), the values for the port type and the operation themselves are fixed for each `invoke` activity and cannot be changed as the process runs.

[1] The Business Process Execution Language for Web Services (BPEL4WS or BPEL) language specification represents the current state of the art in process-oriented Web service composition languages. It is currently undergoing a standardization process at OASIS, which may change some of its capabilities. In this example we refer to version 1.1 [14].

```
<process
 xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
 name="AsyncEchoService" targetNamespace="urn:asyncEcho:Service"
 xmlns:this="urn:asyncEcho:Service" suppressJoinFailure="no"
 enableInstanceCompensation="no" abstractProcess="no">

  <partnerLinks>
     <partnerLink name="caller" partnerLinkType="this:EchoPLT"
      myRole="service" partnerRole="client"/>
  </partnerLinks>

  <variables>
     <variable name="echoMessage" messageType="this:EchoMessage"/>
  </variables>

  <sequence>
     <receive name="echoReceive" partnerLink="caller"
      portType="this:EchoService" operation="echo"
      variable="echoMessage" createInstance="yes"/>

     <assign>
        <copy>
          <from variable="echoMessage" part="replyTo"/>
          <to partnerLink="caller"/>
        </copy>
     </assign>

     <invoke name="echoReply" partnerLink="caller"
             portType="this:EchoClient" operation="echoCallback"
             inputVariable="echoMessage"/>
  </sequence>
```

Fig. 2. Example of dynamic redirection of service end points with BPEL. The `assign` activity copies a service end point reference from the incoming message to the partner link which is going to be invoked afterwards.

Thus, BPEL limits the scope of a binding depending on the time it is evaluated. Although a static binding can refer to different services identified by their port types, a dynamic binding is restricted to switch between the various ports of the service. This limits the reusability of composite services modeled with BPEL. Furthermore, by using WSDL port types to define constraints in the roles that can be assumed by each partner, such constraints are expressed only at the level of the syntactical description of the service interface and no explicit provision for semantics is made [16]. Thus, services which do not match the interface description contained in the binding will not be considered as potential replacements. Conversely, it is not possible to discard services that use the same syntactical signature, but provide incompatible functionality.

Before we present in detail how we address these limitations with JOpera, we introduce the notion of flexible binding abstracting from the details of a specific service composition language.

3 Flexible Binding

In general, a *binding* of a service into a composition can be defined as the reference used to choose the service to be invoked as part of the compostion. As we will discuss in the rest of this section, there are different ways of identifying the services to invoke. Furthermore, a binding can be evaluated at different times during the life cycle of a composition.

3.1 Modeling bindings

Different composition languages may take different approaches to describing how components are bound into the composition. In this paper we make very little assumptions about how a composition language is used to model a composition. In particular, we assume that a composition contains two different kinds of information. 1) A set of bindings $\{b\}$, used to identify which services should be composed. 2) A model of the structural relationship between the bindings.

Although it is outside the scope of this paper to detail such model, it is worth noting that based on the structure of the composition it is possible to define constraints on the services that can be bound into it. For example, if the composition includes information about the data flow dependencies between the services, this information can be used to constrain the services that can be used. On the one hand, the data flow of the composition defines what data each service should be able to produce and consume. On the other hand, if two services are connected, they must fit with one another, i.e., be able to exchange messages with compatible content. As another example, if the composition includes control flow dependencies between the operations to be invoked, only services which support compatible interaction protocols can be used within such composition.

Binding by inclusion In the simplest case, services are statically bound into a composition by inclusion ($b = s$). Thus, the description of a service s is mixed with the description of the structure of the composition. Although this solution already captures the relationship between the composition and its component services, it is too simple in order to effectively model the reuse of either. With it, services can only be reused by duplicating their description in different parts of the compositions. Likewise, it is not straightforward to apply the same composition to invoke different services.

Binding by reference involves using references linking the composition to external descriptions of the services to be invoked. Thus, a binding becomes $b = t \rightarrow s$, where t represents the name referencing a service and s describes the service to be invoked. By using a reference, the service description can be stored separately from the composition. This is an important step which enables to model reuse at the level of the services, as it is now possible to have the same service referenced by more than one binding within more than one composition. However, since each binding uniquely identifies the service to be invoked, the model is not yet powerful enough to reuse compositions, where a composition can be applied to different services without modifying it.

Binding by constraint To support reusable compositions, we extend the previous definition of binding to: $b = t \rightarrow S = \{s : C(s)\}$, where now t represents the reference to a set of services S. Thanks to this approach, a composition may be reused since its bindings only contain the constraint C modeling the requirements that a service should satisfy in order to be included in the composition.

Although this enables to reuse a composition, it remains to be defined how to model such constraints as part of each binding and when to evaluate the binding so that the actual service to be invoked can be determined based on the available known services. Depending on the available services and the actual set of constraints, it may occur that no service can be found as a result of the binding's evaluation. This condition will result in a failure of the execution of the composition. More precisely, issues such as service substitutability [7] and what is the information that should be provided to identify a service and to determine its equivalence with others remain open [11]. This is an important problem, as the services to be bound into a composition must fit within its structure. In other words, not all possible services may comply with the syntax and semantics assumed by the composition as well as by the other services which are part of it.

The problem of modeling these constraints can be addressed at different levels of abstraction. As we have shown in the example of the previous section, depending on the compostion language, it may be possible to define alternative communication end points associated with a given service interface. This way the composition does not contain hard-coded information about specific access paths to the service's functionality, but only defines how one should be chosen. Abstracting from the communication details, a larger number of services could be bound into a reusable composition as long as they have a compatible semantics [6]. Thus, as part of the binding, it should be possible to constrain the interface of the referenced service accordingly (e.g., by using ontologies [2], interface templates [8], service offerings and constraint groups [27], abstract functionalities [17], or formal functional specifications [33]).

3.2 Evaluating bindings

Once a composition includes flexible bindings that do not uniquely identify the services to be composed, the composition system must evaluate such binding $(e(b) : b \rightarrow s \in S)$ so that the evaluation e selects the service s to be invoked among all possible ones (S) that satisfy the binding's constraints. For a composition, the result of the evaluation of all of its bindings forms a *binding configuration*, which describes which services are going to be used for each of its bindings. This evaluation can be influenced by different factors (Figure 3).

First of all, it may be useful to restrict the set of the services targeted by a binding. For example, *blacklisting* is used to guarantee that a set of services will not be used when dereferencing the binding. This mechanism allows to ensure that when an existing composition is reused, for example, untrusted providers, whose services have been added to the blacklist, are excluded from the set of services that can be bound into it.

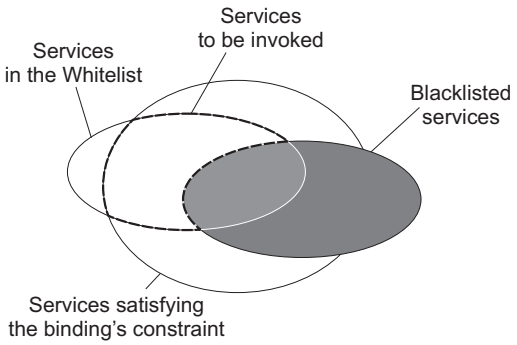

Fig. 3. Evaluating a binding using additional constraints

Conversely, *whitelisting* is used in a complementary way. Whereas a blacklisted service will not be considered, a binding constrained by a whitelist will reference only services that are explicitly enumerated in such list. This way, during the evaluation of the binding it is possible to control that the services invoked by the composition, e.g., belong to a set of services that should be used only for testing purposes.

Formally, the set of services that can be invoked I is related to the original set S defined by the composition's binding as follows:

$$I = (S \cap W) \setminus B$$

where W represents the set of whitelisted services and B the blacklist. Although it is unlikely that a service belongs to both lists at the same time, with this definition the blacklist has the higher priority.

In addition to these two policies that control by exclusion or inclusion what are the services that can be chosen, the choice of the service resulting from the binding's evaluation can also be driven by Quality of Service considerations [18]. In this case a trade off is involved, e.g., each service is selected by minimizing the price (or invocation cost) associated with it, while maximizing the expected performance (e.g., in terms of the guaranteed response time of the service). This kind of metadata is typically maintained by service registries (e.g., UDDI [19]) which use a combination of automatic tools and manual validation to ensure its correctness. However, there is still a lot of work that remains to be done before issues such as *trust establishment* can be dealt with in a fully automatic manner.

3.3 Beyond static and dynamic binding

In the previous section we have presented what are some of the options that can affect the evaluation of a binding, influencing the way a service is chosen to be included in a composition. However, we have not discussed when this evaluation may occur. During the life cycle of a composition, there are many opportunities

for taking such decision. Thus, the traditional distinction between static vs. dynamic (or early vs. late [10]) binding can be refined by differentiating between the following evaluation times.

Registration time First of all, even before a composition is defined, pre-existing services are classified using a registry. The way individual services are catalogued affects how they can be discovered and referenced from a composition [9].

Composition time During the definition of a composition, the developer makes a selection of the services to be invoked. At this stage, it is possible to establish a fixed binding to the exact service which should be invoked (a form of early binding). However, this limits the reusability of the composition, which would have to be modified in order to be used with different services. As an alternative it is also possible to associate some more limited constraints with each binding. These constraints will influence the choice of the service, which is delayed to a later stage.

Compilation time Before a composition can be executed, it is usually compiled from the representation used to model it to a representation optimized for execution. Assuming that the compiler has access to quality of service metadata about services, it can use this information to select which services should be bound into the composition based on different policies [25]. Existing approaching advocating the automatic selection of services based on whether they fit with the composition's structural constraints can also be considered as a form of compile-time binding [3]. Compilation time is also a good opportunity for checking the consistency of the binding's constraints with respect to the available services. Although it is possible that services which satisfy a binding will become available after a composition has been compiled, it may be useful to warn the developer about potential problems due to missing services and unsatisfied bindings.

Deployment time At this point, the compiled service composition is deployed in the execution environment. During deployment the the composition changes of hands, going from the control of the developer to the end user which will manage its execution. In fact, it is the latest opportunity for the developer of the composition to customize it by selecting the specific services that should be bound into it. Thus, as part of the deployment, a developer playing the role of system integrator may select what are the actual service providers to use for the particular installation. This way, a reusable composition can be tailored to use different services each time it is deployed to be executed at a different site taking into account the characteristics of the local environment.

Startup time After deployment, a composition is ready to be executed. Bindings can also be evaluated at the very beginning of its execution. This way, as part of the initialization phase of a specific execution it can be decided what are the services that should be used. More precisely, binding on startup refers to the possibility of further constraining the services to be invoked in a different way each time the composition is run.

One interesting application example of this case concerns the testing of the composition. In this scenario, while a composition is developed, it may not be

possible to invoke production quality services. The services to be composed may not yet be available or it may not be possible to use them for testing the composition, as they belong to a pre-existing system which is already in production. Thus, developers may find it useful to start a testing run of their composition by binding some of its services to stubs which will be invoked for testing purposes only.

Invocation time This case is what is mostly referred to as dynamic binding whereby the decision about which services should be used is delayed until the latest possible time, i.e., when the service is about to be invoked.

Although all of the previously described evaluation times involved a certain degree of manual intervention, in this case, we argue that is too late to do so. In practice, the choice of the service during the binding's evaluation should be fully automated for two reasons. Asking a user operator to manually bind a service invocation to a provider each time a service should be invoked would dramatically affect the execution's performance. Furthermore, it should not be assumed that users monitoring the execution of the composition have complete knowledge about details concerning bindings, which are typically under the purview of the original developers of the composition.

Failed Invocation time A special case of dynamic binding, which we would like to distinguish, concerns the re-evaluation of a binding in case of a failed service invocation. The main purpose of this binding on retry mechanism is to enhance the reliability of the composition by offering the capability of selecting a different service if the one resulting from the first evaluation of the binding turns out – at run-time – to be unavailable. More precisely, the default service referenced by the binding is called as if the binding would have been a static one. If the invocation succeeds, the execution of the composition continues normally. In case of failures due, for example, to the temporary unavailability of the default remote service provider, this mechanism allows to invoke a backup service which – in general – is selected by re-evaluating the binding like in the case described previously.

4 Flexible Binding with JOpera

After giving a general description about flexible binding and how it can be used, we proceed to show how these ideas have influenced the design of JOpera's visual composition language and the corresponding run-time infrastructure. In this section we are argue that a composition language may support flexible bindings without necessarily embodying this notion into a specific language construct. Instead, we will show that different kinds of bindings can be all specified by using reflection.

4.1 Modeling bindings with reflection

As opposed to introducing a specific language feature to support flexible bindings, in JOpera we have taken a more general approach based on *reflection*.

Reflection is the ability of a computational system to represent and modify information about itself [15]. In JOpera, reflection is used to access and modify metadata about the static structure of a composition, its current state of execution, as well as to interact with the services provided by the runtime system [21]. In the first case, the composition language extends the basic service invocation construct with *system parameters* (In the visual syntax, they are shaded in gray and their name is prefixed with `SYS`). Thus, in addition to input and output parameters describing the data which is sent and received from a service, the system parameters allow to control the invocation mechanism and access related metadata. Furthermore, *system services* model the interaction between a composition and the underlying runtime infrastructure.

In the context of this paper, reflection is a mechanism used to expose in the composition language the binding and registry services provided by the runtime environment so that they can be controlled from within a composition. More precisely, we are interested in accessing a registry listing available services and in controlling the way a binding is evaluated.

One of the advantages of reflection is that it leaves ample freedom to model the constraints associated with a binding in many different ways. As we are going to show, with reflection it is possible to distinguish which part of the composition should be dynamically bound from the policy controlling how such binding should be evaluated. In the most advanced case, through reflection, a composition may – for instance – dynamically modify itself to bind to a service whose interface requires some form of adaptation to fit with the rest of the composition.

4.2 Bindings in the JOpera Visual Composition Language

In the rest of this section we illustrate with an example how to use reflection to model different kinds of bindings constraints: fixed bindings, where the constraint determines exactly which service should be invoked; communication level constraints, applied to the communication end points to be used by the services; structural constraints, defining minimal requirements on the syntax of a service interface; but also even how to remove all constraints, where a binding is left completely free so that a composition may call any Web service.

The examples shown in Figures 4 and 5 involves a typical (and reusable) interaction pattern between a client and a service playing the role of broker. More precisely, depending on the client's request, the broker will lookup what are the available supplier services, forward them the original client request and gather their corresponding bids, which are finally sent back to the client. Although this is a simple example, it can be implemented in different ways depending on the required level of reusability of the composition.

Fixed binding This is the simplest form of binding, where a service is statically bound into a composition. In case of the example shown in Figure 4, this form of binding is applied to the `lookup` service. With it, a default `RegistryProvider` is assigned to the lookup service invocation. However, with the tools provided

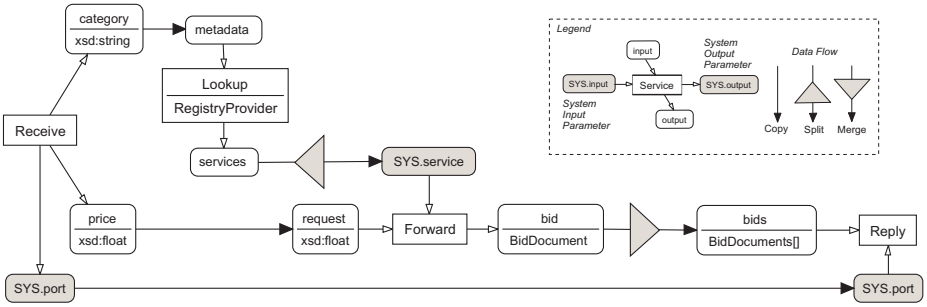

Fig. 4. The first version of the broker composite service using dynamic binding with structural constraints.

with JOpera it is still possible to replace the registry service, both when the composition is deployed as well as each time it is started.

Communication level constraints In the case of services, whose interface is bound to a given provider, it is still possible to dynamically choose the communication port (or end-point, in WSDL terminology) which should be used to communicate with it. In both versions of the example, the `Reply` is constrained to be invoked on the same `port` that was used to perform the `Receive`. In other words, this binding constraint ensures that the answer of the broker composite service goes back to the client which submitted the original request. In general, a similar approach can be used to model constraints related to asynchronous message exchanges so that a composition can be reused to handle the interaction between different services that follow the same conversation [5].

Structural constraints As a general note, this example follows the principle of separating the binding of the service invocation from the strategy used to evaluate the binding. More concretely, the evaluation of a binding can be modeled as the invocation of a lookup operation of a registry service. In this case, reflection is used to expose the registry to which the appropriate query is sent. In addition to JOpera's internal registry, such lookup funcionality can also be provided by an external registry (e.g., UDDI [19]) or search engine (e.g., Woogle [8]). Thus, a binding constraint corresponds to a query to a service registry. Depending on the capabilities of such registry, a query may be based on `metadata` identifying the context that should be used to filter the resulting list of `services`, as shown in Figure 4. Not shown explicitly in the example, the query sent to the `Lookup` service also includes a structural constraint on the interface of the services to be returned. In particular, the composition specifies that candidate services to be bound in place of the `Forward` invocation must comply with its interface, i.e., they must accept at most one parameter (`request`) or a certain data type (a floating point number as prescribed by XML schema [31]). Furthermore, the result of the services must contain at least a parameter named `bid` of type `BidDocument`. Depending on the matching algorithm employed by the particular

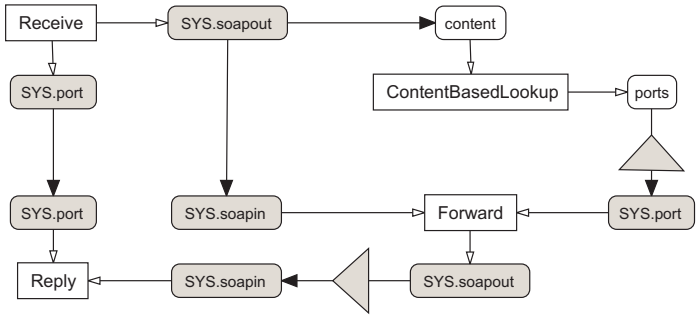

Fig. 5. The second version of the broker composite service using dynamic binding without any constraints.

lookup service, these structural constraints would allow the composition to be applied to services returning additional parameters (which are simply discarded by the composition) and accepting a subset of the required input data parameters, assuming that the service interface defines default values to be used for the missing input parameters.

Unconstrained bindings In the second version of the example shown in Figure 5, the broker composite service can be reused with minimal constraints on the content of the messages that are exchanged between client and suppliers. By using reflection to expose details such as the content of the raw SOAP messages (the `SYS.soapin` and `SYS.soapout` parameters), it is possible to reuse this composition which only describes the interaction patterns between the services involved, abstracting from the specific syntax of their interfaces.

More precisely, this broker composite service uses a `ContentBasedLookup` service, which takes the content of a SOAP message to query a registry for compatible services that may be able to consume it. As opposed to the previous example, in order to identify such services, the registry returns the URL to the `WSDL` document describing their interface, and the names of the actual `service`, `operation`, and `port` that should be used. Thus, in addition to exposing the communication port, the dynamic binding of the `Forward` service invocation selects all of the parameters identifying the service to be called. Furthermore, it is the registry's responsibility to correlate the incoming message with the available services which may be able to process it. In the previous example, the extraction of the metadata to be used to find matching services was done as part of the receipt of the message.

This example shows that it is possible to use reflection to access the internal representation of a Web service invocation in order to express a binding which is left completely free to be evaluated at the latest possible time. Clearly, the example is a bit extreme, as all assumptions about the syntax of the data received and produced by the services have been removed from the composition. Thus, its expressiveness suffers as it is impossible to verify (neither statically

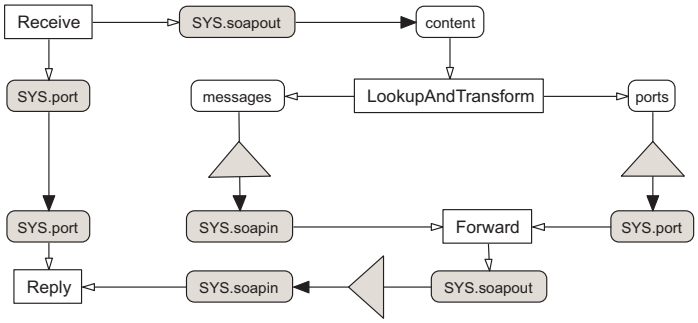

Fig. 6. The third version of the broker composite service can adapt messages before forwarding them to the chosen service end points.

or dynamically) that a service complies with the constraints that are associated with its interface simply because such information is not included in the composition. However, there may be cases for which this level of flexibility is clearly the intended behavior. For these cases, the example of Figure 5 shows how to use a `ContentBasedLookup` service to dynamically find a service matching with a message that is going to be forwarded to it. In a similar way (Figure 6), it is possible to extend the registry service with mediation functionality so that the message used to lookup matching services can be transformed to be consumed by the selected service, in case it cannot be forwarded directly.

5 Related Work

The need for flexibile bindings was recently brought forward in [13], with the argument that the requirements of pervasive computing would challenge current component-based software engineering methodologies. In the context of Web service composition, this challenge has been addressed in different ways, mostly by including specific constructs in the corresponding composition languages. In BPEL [14], as we have exemplified in Section 2, communication end-points can be reassigned at runtime. Although this gives some limited flexibility, it does not enhance the reusability of a composition, as all information about the *port types* and *operations* that identify the Web service to be invoked are bound to constant values. This is not the case in other languages, e.g., XL [12], where flexible binding is supported by letting the argument of an invoke command represent an arbitrary XQuery expression, which is evaluated at runtime to choose the actual service to be invoked.

Reusability of Web service compositions is also the focus of [17], where composition "patterns" are modeled in terms of abstract "functionalities". Following this approach, the actual Web services come into play at "pattern-specialization time", when developers manually select the services which match the functionalities required by the composition. In the same paper the important trade-off

between the abstraction (i.e., potential reusability) of a pattern and its expressiveness is identified. The example of Figure 5 can be interpreted along the same lines. Given the lack of assumptions made by the composition about its components, it is true that the composition can be reused with many services. However, the expressiveness of the composition suffers, as no constraints are given in order to choose such services.

6 Conclusion

In this paper we have presented a different perspective on how to reuse Web service compositions. Not only can compositions of Web services be published themselves as a Web service, but it should be possible to apply the same composition to coordinate different but compatible services.

To do so, we introduced the notion of flexible binding describing the relationship between a composition and its components. In order to enable reusable compositions, a binding should be modeled in terms of constraints. These identify a set of candidate services from which one will be chosen to be invoked after the evaluation of the corresponding binding. Such constraints can be expressed in many different ways: for example, queries over classification meta-data or quality of service information, requirements about the syntax and semantics of service interfaces, as well as blacklisting and whitelisting of service providers. Furthermore a binding can be evaluated at different times during the lifecycle of a composition, going beyond the classic distinction between early and late binding, we have exemplified several different points in time (registration, composition, compilation, deployment, startup, invocation and retry on failure) in which a binding may be fully evaluated or more constraints can be added to it. Thanks to the flexible binding linking the description of the composition to the description of the components, which are kept separate, not only compositions become reusable but their reliability and testability may be improved.

Given the wide range of different approaches to modeling bindings that have been introduced in existing composition languages and tools, but also considering the partial support for flexible binding of the state-of-the-art BPEL4WS language and related tools, in this paper we suggest a different approach based on reflection. With it, it is not necessary to include explicit constructs in a composition language to model flexible binding. Instead, flexible binding can be considered as a particular application of reflection, whereby – as we have shown with several examples – parts of the underlying composition infrastructure are exposed from within the composition language. One advantage of this approach is that it is possible to distinguish how a service is bound into a composition from the strategy used to evaluate such binding.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments, Tone Hansen for pointing out blacklisting as a good strategy from the system integrator's perspective, and Biörn Biörnstad for his expertise with BPEL.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services: Concepts, Architectures and Applications*. Springer, November 2003.
2. A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. McDermott, D. Martin, S. A. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. DAML-S: Web Service Description for the Semantic Web. In I. Horrocks and J. Hendler, editors, *Proceedings of the 1st International Semantic Web Conference (ISWC2002)*, volume 2342, pages 348–363, Sardinia, Italy, June 2002.
3. D. Berardi, D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of e-Services that Export their Behavior. In *Proceedings of the 1st International Conference on Service-Oriented Computing (ICSOC 2003)*, volume 2910 of *LNCS*, pages 43–58, Trento, Italy, December 2003. Springer.
4. A. D. Birrel and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.
5. M. Brambilla, S. Ceri, M. Passamani, and A. Riccio. Managing Asynchronous Web Services Interaction. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, pages 80–88, San Diego, California, June 2004.
6. C. Bussler. Semantic Web services: Reflections on Web Service Mediation and Composition. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE 2003)*, pages 253–260, Roma, Italy, December 2003.
7. V. De Antonellis, M. Melchiori, B. Pernici, and P. Plebani. A Methodology for e-Service Substitutability in a Virtual District Environment. In *Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE 2003)*, pages 552–567, Klagenfurt, Austria, 2003.
8. X. Dong, A. Y. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Simlarity Search for Web Services. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, pages 372–383, Toronto, CA, August 2004.
9. V. Draluk. Discovering Web services: An Overview. In *Proceedings of 27th International Conference on Very Large Data Bases (VLDB 2001)*, pages 637–640, Roma, Italy, 2001.
10. M. Elson. *Concepts of Programming Languages*. Scientific Research Associates, Chicago, 1973.
11. D. Fensel and C. Bussler. The Web Service Modeling Framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, Summer 2002.
12. D. Florescu, A. Gruenhagen, and D. Kossmann. XL: An XML Programming Language for Web Service Specification and Composition. In *Proceedings of the 11th international conference on World Wide Web (WWW'02)*, pages 65–76, Honolulu, Hawaii, USA, 2002.
13. T. Gschwind, M. Jazayeri, and J. Oberleitner. Pervasive Challenges for Software Components. In *Proceedings of the 9th International Workshop on Radical Innovations of Software and Systems Engineering in the Future (RISSEF 2002)*, pages 152–166, Venice, Italy, October 2002.
14. IBM, Microsoft, and BEA Systems. *Business Process Execution Language for Web services (BPEL4WS) 1.0*, August 2002. `http://www.ibm.com/developerworks/library/ws-bpel`.
15. P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 147–155, Orlando, FL, October 1987.

16. D. J. Mandell and S. A. McIlraith. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In G. Goos, J. Hartmanis, and J. van Leeuwen, editors, *Proceedings of the Second International Semantic Web Conference (ISWC2003)*, volume 2870 of *LNCS*, pages 227–241, Sanibel Island, Florida, 2003. Springer.
17. L. Melloul and A. Fox. Reusable Functional Composition Patterns for Web Services. In *Proceedings of the Second International Conference on Web Services (ICWS2004)*, pages 498–505, July 2004.
18. D. A. Menascé. Composing Web Services: A QoS View. *IEEE Internet Computing*, 8(6):88–90, November-December 2004.
19. Oasis. *Universal Description, Discovery and Integration of Web services (UDDI) Version 3.0*, 2002. `http://uddi.org/pubs/uddi_v3.htm`.
20. C. Pautasso. JOpera: Process Support for more than Web services. `http://www.iks.ethz.ch/jopera/download`.
21. C. Pautasso. *A Flexible System for Visual Service Composition*. PhD thesis, Diss. ETH Nr. 15608, July 2004.
22. C. Pautasso and G. Alonso. Visual Composition of Web Services. In *Proceedings of the 2003 IEEE International Symposium on Human-Centric Computing Languages and Environments (HCC 2003)*, pages 92–99, Auckland, New Zealand, 2003.
23. C. Pautasso and G. Alonso. From Web Service Composition to Megaprogramming. In *Proceedings of the 5th VLDB Workshop on Technologies for E-Services (TES-04)*, Toronto, Canada, August 2004.
24. C. Pelzu. Web Services Orchestration and Choreography. *COMPUTER*, 36(10):46–52, October 2003.
25. S. Ran. A framework for discovering Web services with desired quality of services attributes. In *Proc. of the 1st International Conference on Web Services (ICWS 2003)*, pages 208–213, Las Vegas, 2003.
26. C. Szyperski. Component technology - what, where, and how? In *Proceedings of the 25th International Conference on Software Engineering*, pages 684–693, Portland, Oregon, USA, 2003.
27. V. Tosic, K. Patel, and B. Pagurek. Reusability Constructs in the Web Service Offerings Language (WSOL). In *Proceedings of the Second International Workshop on Web Services, E-Business and the Semantic Web (WES 2003)*, volume 3095 of *LNCS*, pages 105–119. Springer, 2004.
28. W. Tracz. *Confessions of a Used Program Salesman*. Addison-Wesley, 1995.
29. W3C. *Simple Object Access Protocol (SOAP) 1.1*, 2000. `http://www.w3.org/TR/SOAP`.
30. W3C. *Web services Definition Language (WSDL) 1.1*, 2001. `http://www.w3.org/TR/wsdl`.
31. W3C. XML Schema, 2001. `http://www.w3.org/TR/xmlschema-0/`.
32. W3C. *Web Services Addressing (WS-Addressing)*, 2004. `http://www.w3.org/Submission/ws-addressing/`.
33. A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(4):333–369, October 1997.
34. L.-J. Zhang and M. Jeckle. The Next Big Thing: Web services Collaboration. In *Proceedings of the International Conference on Web services (ICWS-Europe 2003)*, pages 1–10, Erfurt, Germany, 2003.