WS Binder: a Framework to enable Dynamic Binding of Composite Web Services

Massimiliano Di Penta[1], Raffaele Esposito[1], Maria Luisa Villani[1]
Roberto Codato[2], Massimiliano Colombo[2] and Elisabetta Di Nitto[2]
dipenta@unisannio.it, r.esposito@unisannio.it, villani@unisannio.it
roberto.codato@cefriel.it, mcolombo@cefriel.it, dinitto@elet.polimi.it

[1] RCOST University of Sannio
Via Traiano, 82100 Benevento, Italy

[2] CEFRIEL – Politecnico di Milano
Via Fucini 2, 20133 Milano, Italy

ABSTRACT

The rapid diffusion of service–oriented systems is becoming a reality in today's software engineering. In particular, an aspect that is gathering the interest of researchers and practitioners is the possibility to create compositions of dynamically bound services. This paper describes WS Binder, a framework for enabling dynamic binding of service compositions according to some functional and non–functional preferences and/or constraints. The framework is also able to support run–time recovery actions, by performing service re–binding. The paper describes the framework's architecture and highlights its features by describing an example of its usage for the binding and re–binding of a service composition related to the tourism domain.

Categories and Subject Descriptors

D.2 [**Software Engineering**]

General Terms

Domain Specific architectures, Data mapping, Distributed objects

1. INTRODUCTION

Web services and service–oriented architectures can be considered among the most promising and challenging technologies introduced during the last few years. Their diffusion and adoption can rapidly change the landscape of the existing software engineering approaches.

Among the various advantages introduced by the service oriented paradigm, it is worth mentioning run–time service discovery and late-binding. Run time discovery enables the automatic retrieval of services by matching a semantic request to service descriptions. It may happen that, in correspondence of a service request (hereby referred to as *abstract service*), several services (hereby referred to as *concrete services*) are found offering the same piece of functionality (e.g., booking a hotel or returning a city temperature). The choice between these services can be dictated by either functional properties (typically, a service performing some additional features is preferred. In the tourism domain this could be a hotel booking service also providing support for booking excursions), or non–functional properties, referred to as Quality of Service (QoS) attributes. For instance, one may decide to choose the cheapest service, the fastest, or maybe a compromise between the two. Moreover, the choice could be influenced by some predefined constraints on the values of some other attributes (e.g., service availability should be greater than a given value).

Given the possibility of choosing among various services, when dealing with a service composition (either realized using traditional programming languages or orchestration languages such as BPEL4WS [2]) designers need to determine the set of concretizations (i.e., bindings between abstract and concrete services) that not only fulfill the functional requirements, but also satisfy the QoS constraints imposed by the Service Level Agreement (SLA) defined with the service user. Finding the solution of such a problem is NP–hard and requires the adoption of some optimization technique.

Moreover, at execution time, it may be necessary to change the bindings determined before execution, for different reasons:

1. initial bindings are based on estimates of services QoS and on assumptions on the paths executed in the composition workflow while these hypotheses can result to be false at runtime;

2. new services can be published that better fit the composition and its QoS constraints; or

3. some services may not be available anymore.

Also, in some cases it is more appropriate not to fix any binding before execution since, for instance, this depends on the context in which the composition is used and this context cannot be foreseen before execution. For example, the concrete service used to send directions to a traveler

IW-SOSE'06, May 27–28, 2006, Shanghai, China.

about the path to be followed has to be decided on the basis of the device the user can exploit at that time and on his/her preferences. In this case, the binding can be determined directly at run–time, based on local preferences/constraints and on the actual availability of services.

This paper describes WS Binder, a framework for dynamic binding of service compositions implemented on top of a BPEL technology. The framework supports pre–execution binding (able to satisfy global composition constraints), run–time binding of single service invocations and run–time re–binding of the whole workflow[1]. WS Binder is part of a larger platform for the development and execution of service–centric systems, that also includes components for service specification, discovery, composition, publication, monitoring and testing. Such a platform is being developed within the SeCSE (Service–Centric System Engineering) European IP Project[2].

The remainder of the paper is organized as follows. Section 2 describes the main features of WS Binder and summarizes the binding and re–binding approaches. Section 3 provides details about the WS Binder architecture, while Section 4 describes WS Binder at work for the binding and run–time re–binding of a travel planning process. Section 5 relates our work to the existing literature, and Section 6 concludes.

2. BINDING AND MAPPING MECHANISMS

WS Binder has been designed to be as flexible as possible both in terms of the point in time when binding is actually performed and in terms of the functional and non-functional policies that can be used to select a service to bind. In the following of this section we describe in detail the features proposed by the system. We rely on the existence of an external discovery service[3]. A service of this kind is currently under development within the SeCSE project.

2.1 Binding mechanisms

2.1.1 Pre–execution workflow global binding

The global binding is performed before the actual execution of a composition and aims at determining the (sub)optimal set of bindings that satisfies functional and non-functional constraints and optimizes an objective function. To this aim, it is necessary to estimate the workflow QoS from those of the potentially bound services and from information about the workflow itself (e.g., the probability that a branch in a switch statement will be executed or the average number of iterations for a loop – these parameters can be either declared by the workflow designer or they can be estimated, after the composite service is working, from monitoring data). In doing so we use aggregation formulae similar to those defined by Cardoso [6]. Essentially, these formulae are defined for each pair workflow construct-QoS attribute. For example, the response time of a loop node will be equal to the response time of the internal nodes times the estimated number of loop iterations. Instead, the response time of a switch will be a weighted sum of the response times of the case nodes with respect to their probability to be executed. Of course, alternative aggregation formulae could be applied in this case, like worst case response time.

To identify the sub–optimal solution we exploit Genetic Algorithms (GAs) that have demonstrated to handle non–linear aggregation formulae, and to be more scalable than other techniques such as Integer Programming (see the paper [4] for details). The genome is represented by an integer array with a number of items equals to the number of distinct abstract services present in the process specification. Each array item, in turn, contains an index to the array of the services matching that abstract service. Although a better solution could probably be reached by considering service invocations as variables, rather then services, this could have an impact on the performance of the optimization algorithm, as discussed in [3]. The two-points crossover and a mutation operator that randomly changes a binding are used to generate new individuals. In this generation process towards convergence, the individuals with the best value of the fitness function will reproduce. The fitness function for a genome g is:

$$F(g) = \sum_{i=1}^{n} (w_i \cdot V_i(g)) + w_d \; D(g) \qquad (1)$$

where $V_i(g)$ is a normalized value, in the interval $[0, 1)$, of one of the n QoS attributes considered, evaluated on the workflow. Note that attributes are normalized (see Zeng *et al.* [10] for details) so that the $V_i(g)$ have to be maximized. In other words, also penalty attributes (e.g., response time or price) are normalized so that higher values of $V_i(g)$ correspond to better QoS (i.e., smaller response time or price).

Each w_i in (1) is a real, positive weight accounting for the importance a service integrator (or user) gives to the $i-th$ QoS attribute. Finally, $D(g)$ is the distance of the fitness value from the constraint, and w_d weights the penalty factor. Once a solution to the composition optimization problem is found, the initial set of bindings is established and the service may be executed.

2.1.2 Run–time local binding

Run time local binding permits the selection of a service binding while the composition is running, just before the corresponding abstract service is invoked. This approach can be seen as complementary to the pre–execution global binding and offers the following advantages: i) it allows to consider only the services available during execution (that may or may not be the same available before execution, especially if the workflow execution time is particularly long), ii) it can be used to base the selection of a service on context information that is available only at run–time, and iii) it determines a binding only for the abstract services that will be actually executed (this may be particularly advantageous over the pre–execution global binding approach when various branches in the composition exist that are executed only rarely). The reader can easily understand that the binding overhead in this case is entirely charged on the execution of the composition. Indeed, this overhead in some cases is not avoidable (no a-priori knowledge of available services) and it is compensated by an increase in the flexibility of the composition.

In general, the local selection can be based on functio-

[1]To be more precise, of the slice still to be executed, as detailed in Section 2.1.3.

[2]*http://secse.eng.it*

[3]A thorough description of the discovery approach is out of scope of this paper.

nal/QoS properties defined for the particular abstract service of the composition, and performed by a selection algorithm that can be indicated in the service binding preferences. In fact, the environment allows to plugin new selection algorithms/policies, so to be able to choose the one to be used, for example according to the required performance or precision. In case of a QoS-based selection, each candidate concrete service for the binding would receive a score that accounts for all the QoS properties specified. For example, this score can be a weighted sum of the values for each QoS attribute of interest. The service selected will be the one with the highest score.

2.1.3 Run–time workflow slice re–binding

Workflow slice re–binding occurs when a service is not available or when measured QoS values deviate from the estimates and would cause a constraint violation. In fact, this could happen because the actual number of times a loop is iterated is much different from the estimation, or if the branch with low probability is actually taken in a conditional node, or the QoS values measured when a service is invoked are not the ones upon which the initial concretization was based. To this aim, the workflow execution is constantly monitored in order to update the QoS estimates on the run and trigger re-binding if required.

In these cases, the execution of the composition is stopped, the workflow slice still to be executed is determined and the global binding approach of Section 2.1.1 is executed again.

After selecting new service bindings, the execution can be resumed. In case the re–binding fails, the framework can be configured (using the preferences settings) to perform different actions such as i) proceed anyway (if possible); ii) suspend the execution; or iii) terminate it. Further details on the re–binding mechanism can be found in [5].

2.2 Service interface mapping

In some cases syntactic differences between a service request and the operation offered by a service make the binding between the two impossible. The goal of the mapping mechanism is to provide some simple adaptation mechanisms that transform the request into the format expected by the service operation and do the opposite transformation on the result (if any). In order to achieve this, the concrete services must refer to an interface richer than a simple WSDL. The current version of the WS Binder considers services described through a set of *facets*, where each facet describes a specific aspect of a service (for instance, we envisage the behavioral facet that can describe the behavior of a service in terms of a state machine describing it). In particular, for the mapping purpose, a semantic facet is assumed to map the input and output parameters of a WSDL interface to a specific ontology. A service integrator willing to exploit the mapping feature must define, in a similar way, the correspondence of the input and output parameters of an invoke activity to the concepts defined in the same ontology.

At runtime, the mapping is performed by components called *adapters*. In general a handler applies a deterministic mapping looking for concepts match. By navigating through the ontology, adapters are able to actually map related concepts. The underlying ontology can be extended by introducing new application-specific concepts.

There could be situations where concept mapping is not enough. For instance, if more than one concept in the source parameters (i.e., parameters of the abstract service request) matches a concept in the destination parameters (i.e., the parameters of the concrete service), the service mapping could not be straightforward. For example, name and surname of a person could be in the same parameter in one case but distinct parameters in other cases. Service integrators can then define ad-hoc adapters that are able to address typical situations. These user-defined adapters can be easily plugged into the system.

2.3 WS Binder management capabilities

2.3.1 Setting up the preferences

The framework binding capabilities can be customized by expressing preferences through a proper configuration interface. More in detail, the aspects that a service integrator can configure are the following:

- Set up global QoS objectives, i.e., decide which QoS attribute(s) should be maximized and which minimized.
- Define workflow-wide constraints (e.g., the response time of the whole workflow should be less than 10 s). A global constraint is defined by: a QoS property (identified by a name), a relational operator, a value, and a weight. Global constraints are intended to be in *and* among each other.
- Fix some local preferences in correspondence to an abstract service of the workflow (e.g., a QoS local constraint stating that the *book hotel* abstract service must be bound to a concrete service that costs less than 10 dollars).
- Identify dependencies between services (e.g., if I use a hotel service to book a stay, I would book the airport transfer using a shuttle company having special rates for that hotel).
- Explicitly include/exclude some services (e.g., I would avoid services provided by a competitor) among those available for binding.
- Define preferences related to re–binding (see Section 2.1.3). In particular, it is possible to define when a workflow re–binding is needed and what action should be performed if the re–binding fails (e.g., no alternative is found to replace a unavailable service, or a timing constraint has been already violated).

2.3.2 Binding management interface

WS Binder offers a management interface that allows to display the following data:

- The paths followed while executing composite service workflows.
- The set of bindings determined both at pre–execution and at run–time.
- The new bindings in case a re-binding action has been performed.

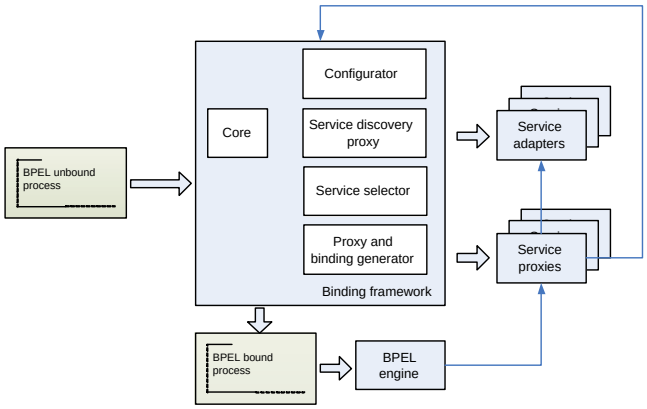


Figure 1: The WS Binder architecture

- Graphs visualizing QoS values before execution (estimated), before re-binding (i.e., the value that has caused the re–binding), after re–binding and when the execution is complete. Examples of these graphs are shown in Section 4.
- Information about the availability of services used during the execution.

All execution data are obtained through a monitoring mechanism that is not part of WS Binder but interacts with it.

3. WS BINDER ARCHITECTURE

WS Binder (see Figure 1) operates at two steps of a composition lifecycle, just before the execution of a composition and during the execution itself. In the first case the framework is in charge of determining the set of bindings that optimizes the quality of service of the entire composition. During the execution, instead, WS Binder aims at dynamically binding the services either based on a global or a local (re)–binding policy.

3.1 Pre-execution binding

At pre-execution time WS binder takes as an input a BPEL workflow where the bindings to services are left unspecified (partner links are not bound to concrete services) and a set of configuration preferences. While processing this workflow, the framework goes through the following steps:

1. *Configuration:* it reads the configuration preferences and configures its state based on them (e.g., minimize the response time and keep the cost within 10 dollars).
2. *Service discovery:* it exploits the functionality offered by a discovery tool in order to obtain a list of services that can fulfill all the required operations. These services can either offer interfaces that are syntactically equivalent to the required ones or can offer some variations of them. In this last case, at runtime, any selected variation will have to be mapped into the corresponding required interface as we have discussed in Section 2.2.
3. *Service selection:* it applies our approach to select, among the proposed ones, the services that best fulfill the QoS requirements.

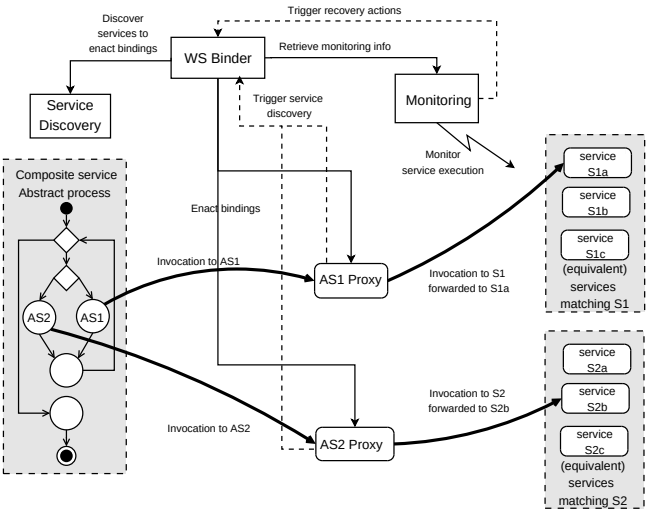


Figure 2: Enacting bindings through proxy services

4. *Service binding and instantiation of proper proxies:* once the concrete service to be bound to an abstract service has been chosen, the binder instantiates a proper proxy service and initializes it so that all incoming invocations will be forwarded to the appropriate service.

The final output of this phase is a BPEL workflow where all service bindings appear to be specified. Technically, these will not point to the actual component services. Instead, they will refer to the proxies generated in this phase (see Figure 2). These proxies will mask the actual invocation of services and will take care of re–binding if needed. In case the configuration preferences specify that pre-execution binding is not required, WS Binder will skip the discovery and selection phases, but it will still generate the proxies that will be in charge of managing the runtime binding and re–binding.

3.2 Runtime (re)binding

At runtime the proxies manage the invocation of the actual services. If no binding has been specified in the previous phase or if a failure or any other problem arises, WS Binder comes into place by selecting, based on the configuration preferences, the slice of the workflow to be (re)bound. This can either consist of a single service or of the workflow fragment that leads from the current point to the termination of the workflow execution. In both cases the steps of discovery and selection are performed on the selected slice. The policy applied in both cases depends on the rebinding preferences defined by the service integrator. When the selection is completed, all proxies involved in the change are provided with bindings to the new services and the control passes back to the proxy that initially called WS Binder so that it can complete its operation.

In case a bound service is not syntactically equivalent to the requested service, the proxy calls a service adapter that applies the strategies we have presented in Section 2.2.

4. WS BINDER AT WORK

This section demonstrates the features of WS Binder by

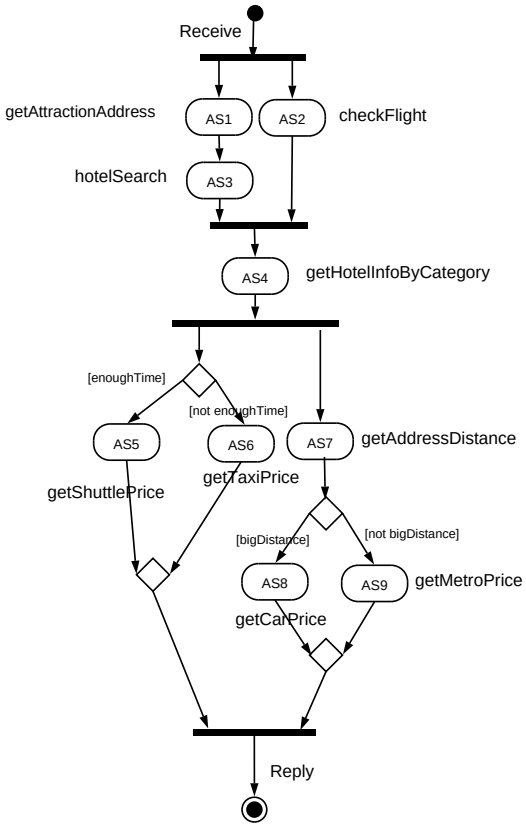


Figure 3: Travel plan process

referring to a travel planning example. It is developed as a composition of other services as shown in Figure 3. In particular, travel planner allows to search for (i) a flight from a starting location to a specified city; and (ii) the nearest possible hotel to a specified tourist attraction in that city. Depending on the arrival time of the flight and the latest possible hotel check-in time, the service will also check for the cost and availability of a cab or a shuttle. At the same time, depending on the distance between the hotel and the attraction, the service will get the cost of a metro card or of a car rental. The whole travel plan and its costs are then presented to the customer for approval and booking.

According to preliminary executions of the process, the arrival time to destination is such that the probability not to take a cab is 80%, and the hotel is expected to be found in the same area as the tourist attraction with a probability of 90%. During the global binding, these estimates permit to compute the global workflow QoS. Also, for this process, the service integrator has decided to ensure a global response time at most of 45 s and to require that both response time and price are minimized. Figure 4 shows the QoS preferences and the re-binding management decisions taken by the service integrator.

Let us consider two possible execution scenarios:

1. Bob intends to fly from Rome to Paris to attend an important dinner at *Maxim*, and so he can leave at any time during the day. A flight is found easily on the required date as well as a hotel to stay, after dinner, in the city center.

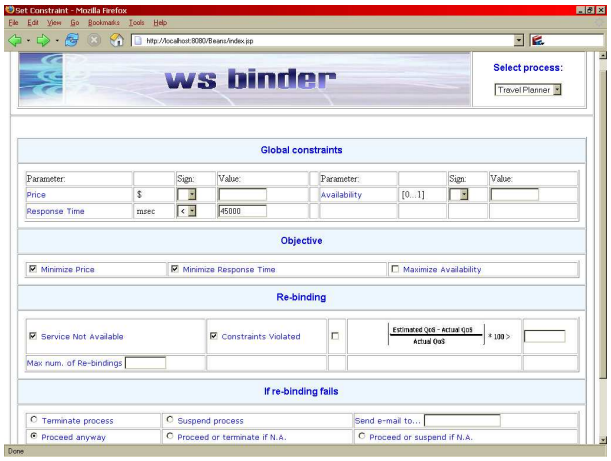
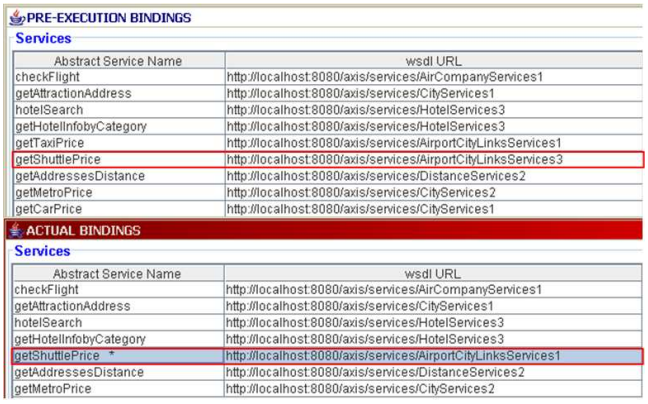


Figure 4: Binding preferences as set by the service integrator



PRE-EXECUTION BINDINGS

Services

Abstract Service Name	wsdl URL
checkFlight	http://localhost:8080/axis/services/AirCompanyServices1
getAttractionAddress	http://localhost:8080/axis/services/CityServices1
hotelSearch	http://localhost:8080/axis/services/HotelServices3
getHotelInfobyCategory	http://localhost:8080/axis/services/HotelServices3
getTaxiPrice	http://localhost:8080/axis/services/AirportCityLinksServices1
getShuttlePrice	http://localhost:8080/axis/services/AirportCityLinksServices3
getAddressesDistance	http://localhost:8080/axis/services/DistanceServices2
getMetroPrice	http://localhost:8080/axis/services/CityServices2
getCarPrice	http://localhost:8080/axis/services/CityServices1

ACTUAL BINDINGS

Services

Abstract Service Name	wsdl URL
checkFlight	http://localhost:8080/axis/services/AirCompanyServices1
getAttractionAddress	http://localhost:8080/axis/services/CityServices1
hotelSearch	http://localhost:8080/axis/services/HotelServices3
getHotelInfobyCategory	http://localhost:8080/axis/services/HotelServices3
getShuttlePrice *	http://localhost:8080/axis/services/AirportCityLinksServices1
getAddressesDistance	http://localhost:8080/axis/services/DistanceServices2
getMetroPrice	http://localhost:8080/axis/services/CityServices2

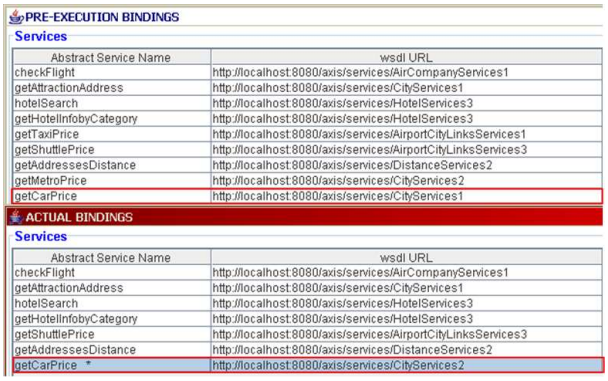
Figure 5: Bob's scenario: shuttle service *AirportCityLinksServices3* not available at run-time

2. Alice intends to fly from Rome to Paris to visit this city for the first time, and she indicates Versailles as the main attraction. Unfortunately, there is not much availability of hotels close to Versailles, thus a hotel is found far away. This leads to suggesting a car rental, with additional pricing information.

Figure 5 illustrates the result to Bob's request in case of re–binding to recover from the unavailability of the shuttle pricing information service initially chosen (*AirportCityLinksServices3*). Thus, *getShuttlePrice* is re–bound to *AirportCityLinksServices1*.

Figure 6.a shows the outcome of Alice's request, where re-binding of *getCarPrice* from *CityServices1* to *CityServices2* is due to the unforeseen big distance (from the process perspective) of the hotel from the attraction indicated. Here the assumption is that all concrete instances of CityService provide both *getMetroPrice* and *getCarPrice* operations.

When performing the initial binding, the binder estimates (using workflow annotations and information collected from monitoring previous executions) that the car rental is unlikely to be needed. Thus, *getCarPrice* is bound to *City-*



PRE-EXECUTION BINDINGS

Services

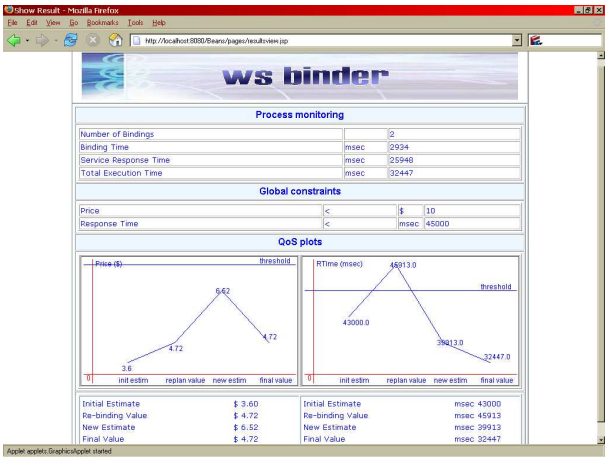
Abstract Service Name	wsdl URL
checkFlight	http://localhost:8080/axis/services/AirCompanyServices1
getAttractionAddress	http://localhost:8080/axis/services/CityServices1
hotelSearch	http://localhost:8080/axis/services/HotelServices3
getHotelInfobyCategory	http://localhost:8080/axis/services/HotelServices3
getTaxiPrice	http://localhost:8080/axis/services/AirportCityLinksServices1
getShuttlePrice	http://localhost:8080/axis/services/AirportCityLinksServices3
getAddressesDistance	http://localhost:8080/axis/services/DistanceServices2
getMetroPrice	http://localhost:8080/axis/services/CityServices2
getCarPrice	http://localhost:8080/axis/services/CityServices1

ACTUAL BINDINGS

Services

Abstract Service Name	wsdl URL
checkFlight	http://localhost:8080/axis/services/AirCompanyServices1
getAttractionAddress	http://localhost:8080/axis/services/CityServices1
hotelSearch	http://localhost:8080/axis/services/HotelServices3
getHotelInfobyCategory	http://localhost:8080/axis/services/HotelServices3
getShuttlePrice	http://localhost:8080/axis/services/AirportCityLinksServices3
getAddressesDistance	http://localhost:8080/axis/services/DistanceServices2
getCarPrice *	http://localhost:8080/axis/services/CityServices2

(a) Pre-execution and actual bindings



(b) QoS values

Figure 6: Alice's scenario: re–binding due to unforeseen distance (need to rent a car)

Services1 since it guarantees a better QoS for *getMetroPrice* (that is expected to be executed). At run–time the process actually needs to execute *getCarPrice*; before its execution, the QoS is re–estimated, a potential constraint violation on the response time is noticed and re–binding is triggered and performed, selecting this time *CityServices2*.

The operation actually offered by *CityServices2* is named *getCarRentalPrice* and requires as parameters the name of the city of rental, the date of rental, and the brand and name of the requested car. Its signature differs from the one of *getCarPrice* both for the name and for the last parameter. *getCarPrice*, in fact, expects as parameter the category of the car to be rent. Thus, a service adapter is required. It will browse the ontology to select a specific brand and type of car belonging to the category specified in *getCarPrice*.

A screenshot of the monitoring view is reported in Figure 6.b, indicating:

- *Summary information about the process binding and execution:* number of bindings (1 if no re–binding was made), binding overhead, execution time of the whole process, sum of response times of the invoked services.

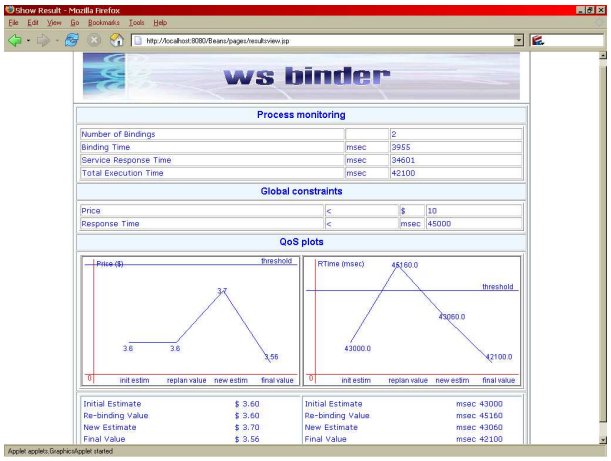


Figure 7: Alice's scenario: re–binding due to response time variation on *getAttractionAddress*

- *Monitoring of price and response time:* the plots show constraints (horizontal lines) and a segment indicating initial estimates, values that trigger re–binding (i.e., above the constraint line) and final values. It is worth noting that final values may be (like in this case) smaller than initial estimates. In fact, initial estimates can be pessimistic (e.g., the cost of a *switch* is computed as a weighted sum of the branches, while at execution time the cheaper branch is followed, or it happens that a *loop* iterates a number of times smaller than what estimated).

Finally, Figure 7 shows the monitoring related to the re–binding triggered by the fact that the *getHotelInfobyCategory* operation took more time than that expected. Without re–binding, this unexpected response time variation would lead to a constraint violation. The problem was solved by re–binding one of the requests immediately following *getHotelInfobyCategory* i.e., *getAddressDistance*, from a slower (but cheaper) implementation to a quicker (even if more expensive) one.

Recalling the fact that the service integrator has indicated that with very high probabilities the system will avoid the user taking a taxi and booking a hotel far away from the attraction, it is possible to derive that some paths in the workflow are more likely to be followed than the others. Therefore, a service integrator might decide not to bind the entire workflow before the execution, but to leave the less likely paths unbounded. In this scenario, run–time local binding will be performed.

5. RELATED WORK

To support a QoS–aware composition, models and techniques for workflow QoS estimation, optimization and monitoring are currently being studied and developed. In this line, the most significant results, closest to ours, are those by Cardoso et al. [6] and Zeng et al. [10]. Namely, the first propose a mathematical model for workflow QoS computation, described by some metrics aggregation functions which are

defined for time, cost, reliability, and fidelity. The work of Zeng et al. [10] focuses on the problem to identify workflow bindings based on optimum global and local QoS criteria. The global optimization problem is solved through integer programming techniques: when the QoS aggregation formula is not linear, linearization is needed. In their approach, loops are unfolded to obtain a Direct Acyclic Graph (DAG), once estimated the number of iterations, and the binding for each task is decided based on the most frequently executed path containing that task. In our work, all the occurrences of an abstract service in the workflow are bound to the same concrete service, so reducing the number of variables of the objective function. In addition, our approach does not limit the binding on particular workflow paths, but is able to determine bindings on the whole process exploiting monitoring information to aggregate QoS values, and using a slicing mechanism to support re–binding.

The problem of determining a global workflow binding that optimizes some objective function has also been modeled by Yu and Lin in [9] as a Multiple Choice Knapsack Problem. With respect to our approach, their solution is, again, limited to workflow having a DAG structure.

A paper by Aggarwal et al. [1] describes a framework for a constraint–driven service composition using linear integer programming. In addition to that, we are able to handle non–linear QoS aggregation formulae. Moreover, the proposed framework enables run–time re-binding.

Serhani et al. [8] propose a QoS broker-based architecture to support the client in selecting web services based on his/her required QoS. The system supports QoS description during the service publication, and dynamic QoS aware invocations. Their approach, however, is limited to local binding. Finally, in [7] authors propose to include ad-hoc service calls in the process description to enable dynamic service discovery and late binding.

In general, the novelty of our approach with respect to the others existing in the literature is that it offers different kinds of binding mechanisms (pre–execution and run–time (re)–binding) in a coherent framework. It also offers the possibility to configure the policy used to select the actual bound services and provides an adaptation mechanism that allows concrete services that are semantically but not syntactically equivalent to the required services to be actually exploited.

6. CONCLUSIONS

This paper has presented WS Binder, a framework that supports the dynamic binding of composite services. Bindings are determined according to functional policies and to global and local QoS optimization criteria.

Global QoS optimization is performed using an approach based on Genetic Algorithms, that permits the use of non–linear QoS aggregation formulae. Both pre-execution and run–time (re)binding are supported. Work-in-progress aims to integrate the binder in the whole SeCSE service execution framework, and to implement more sophisticated features such as the binding based on customized QoS attributes.

7. ACKNOWLEDGMENTS

This work is framed within the IST European IP *SeCSE (Service Centric Systems Engineering)* http://secse.eng.it, 6th Framework Programme, Contract No. 511680.

8. REFERENCES

[1] R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint driven web service composition in METEOR-S. In *Proc. IEEE International Conference on Services Computing (SCC'04)*, pages 23–30, Shanghai, China, Sept. 2004.

[2] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, S. T. D. Smith, I. Trickovic, and S. Weerawarana. Business process execution language for web services. *http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/*.

[3] G. Canfora, M. Di Penta, R. Esposito, and M. Villani. A lightweight approach for QoS-aware service composition. In *Proc. 2nd International Conference on Service Oriented Computing (ICSOC'04)*, New York, USA, Nov 2004. IBM Technical Report.

[4] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An approach for QoS-aware service composition based on genetic algorithms. In *in Proc. of Genetic and Evolutionary Computation Conference (GECCO 2005)*, Washington, DC, USA, June 2005.

[5] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. QoS-aware replanning of composite web services. In *in Proc. of the 3rd IEEE International Conference on Web Services (ICWS 2005)*, Orlando, FL, USA, July 2005.

[6] J. Cardoso, A. Sheth, J. Miller, J. Arnold, and K. Kochut. Quality of service for workflows and web service processes. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(3):281–308, April 2004.

[7] D. Mandell and S. Mcllraith. Adapting bpel4ws for the semantic web: The bottom-up approach to web service interoperation. In *The SemanticWeb - ISWC 2003*, volume 2870/2003, pages 227–241. Springer-Verlag Lectur Notes In Computer Science, 2003.

[8] M. Serhani, R. Dssouli, A. Hafid, and H. Sahraoui. A QoS broker based architecture for efficient web services selection. In *Proc. International Conference on Web Services (ICWS'05)*, pages 113–120, Orlando, FL, Jul. 2005. IEEE.

[9] T. Yu and K. Lin. Service Selection Algorithms for Composing Complex Services with Multiple QoS Constraints. In *Proc. 3rd International Conference on Service Oriented Computing (ICSOC'05)*, pages 130–143, Amsterdam, The Netherlands, December 2005. Springer.

[10] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5), May 2004.