QoS-Aware Adaptive Service Orchestrations

Gianpaolo Cugola, Leandro Sales Pinto and Giordano Tamburrelli
DeepSe Group @DEI, Politecnico di Milano, Italy
{cugola|pinto|tamburrelli}@elet.polimi.it

***Abstract*—Service Oriented Computing enables distributed applications that *orchestrate* existing services exported by remote providers. This paradigm requires to explicitly handle possible changes that may affect the orchestration. They include changes that impact its functional behavior (e.g., services being retired by their providers), but also changes in the non-functional behavior of the orchestrated services (e.g., an increased execution time). In the past we developed DSOL: it combines a declarative language to model the orchestration with planning mechanisms to decide at run-time the best flow of actions. In this paper we extend DSOL to support QoS attributes and requirements. In particular, we combine the DSOL planning techniques with a linear optimizer to calculate the optimal plan w.r.t. the QoS requirements. Moreover, we leverage the DSOL ability to adapt the orchestration flow at run-time, to further optimize the QoS perceived by the end users depending on the actual situations encountered.**

I. Introduction

Service Oriented Architectures (SOAs) [19] are an important and widely adopted paradigm for building business applications. SOAs refer to software systems built from unassociated, loosely coupled units of functionality (i.e., *services*), which are developed, deployed, and operated by independent providers and composed together to provide new value-added services for end-users. Composing (i.e., *orchestrating*) services exported by remote providers brings new challenges to Software Engineering. First, at design-time, engineers have to effectively design the orchestration by selecting an appropriate set of services such that the final system meets its functional and non-functional requirements. Secondly, at run-time, the orchestration should be able to cope with exceptional situations that result from the interaction with an environment (the external services) controlled and managed by other entities. As a consequence, dependable service compositions must be supported by appropriate orchestration languages and run-time systems providing effective mechanisms to design and execute orchestrations able to cope with unexpected behaviors at runtime, i.e., *adaptable orchestrations*.

In response to these challenges we designed DSOL [9] an infrastructure aimed at supporting design and execution of adaptable orchestrations. DSOL models the orchestration declaratively, in terms of its *goals* and the primitive *actions* potentially available to reach them. At run-time, it uses planning techniques to determine the actual flow of execution, i.e., which actions to execute and in which order. Even in presence of changes in the external services available, its internal Planner automatically builds alternative paths of execution that circumvent the changes, avoiding faults and maximizing the chances to reach the orchestration's goals. In presence of major changes, it allows service architects to easily modify the orchestration model at run-time, e.g., by adding new actions or changing the orchestration goals. This eases the job of building "adaptable orchestrations" in the sense above.

However, unexpected changes in the orchestrated services are not limited to their functional behavior but extend to quality of service (QoS) (e.g., response time, reliability, accuracy, etc.). Indeed, changes in the non-functional profile of services may affect the orchestration ability to satisfy its own QoS requirements. In this paper we address this issue by extending DSOL to support QoS.

Q-DSOL (QoS-aware DSOL) models the QoS attributes of external services as part of the available actions, while the QoS requirements of the whole orchestration are modeled as part of its goals. At orchestration invocation, the Q-DSOL Engine uses linear optimization techniques to search for an optimal service binding that could satisfy the orchestration goals, even in presence of conflicting non-functional requirements. In addition, Q-DSOL supports two forms of run-time adaptation. First, it modifies the initial service binding at to achieve further optimization given the knowledge acquired during execution (e.g., the fact that a service, originally considered non-fully reliable, executed correctly). Second, it leverages the DSOL re-planning techniques to optimize the orchestration QoS in presence of faults, maximizing reliability. These forms of optimization and adaptability applies both to pre-defined QoS metrics (execution time and reliability), but also to user-defined metrics, which allow service architects to express domain specific, non-functional requirements.

The remainder of the paper is organized as follows. Section II illustrates ProgrammableDinner, a real-world service orchestration we use throughout the paper. Section III briefly illustrates DSOL's main features, while Section IV discusses how we extended DSOL to support QoS, illustrating the optimization and adaptive mechanisms offered by Q-DSOL. Section V validates our proposal by providing some experimental results of Q-DSOL. Finally, Section VI discusses related work, while Section VII draws some concluding remarks.

II. Programmable Dinner

ProgrammableDinner (PD) is the service orchestration we use to illustrate the functionalities of Q-DSOL. PD orchestrates external services to organize a social event by choosing a restaurant, a movie, and inviting a group of friends. We considered the following requirements:

- $RQ1$: The system shall initially ask the user to provide the relevant data: the list of friends she wants to invite, the

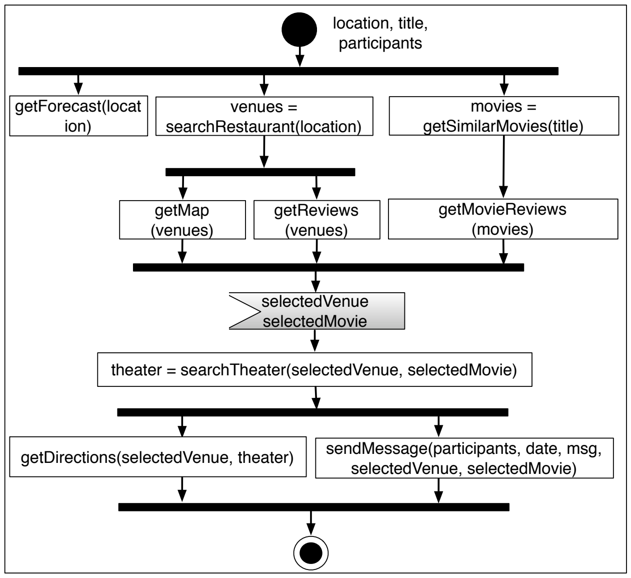

location, title, participants
getForecast(location)
venues = searchRestaurant(location)
movies = getSimilarMovies(title)
getMap (venues)
getReviews (venues)
getMovieReviews (movies)
selectedVenue selectedMovie
theater = searchTheater(selectedVenue, selectedMovie)
getDirections(selectedVenue, theater)
sendMessage(participants, date, msg, selectedVenue, selectedMovie)

Fig. 1. ProgrammableDinner Orchestration Schema

location and a movie title used by the system to suggest similar movies to watch.

- $RQ2$: Based on the indicated location, the system shall provide the weather forecast for the following days, plus a list of restaurants (including a set of reviews about each of them) and a map showing their position. Based on a movie title, the system shall suggest a list of movies similar to the one indicated, including a set of reviews about each of them.
- $RQ3$: Given the data presented to the user, she selects a movie and a restaurant. The system shall provide the list of theaters that play the movie at the requested date.
- $RQ4$: Based on the location of the selected restaurant and theater, the system shall present the needed directions.
- $RQ5$: The indicated list of participants shall receive a message indicating all the details concerning the organized event.
- $RQ6$: The overall system response time, not including the user think time, shall be less than 1.5s and the overall system reliability should be greater than 0.99.
- $RQ7$: The system should perform as fast as possible.

$RQ1$-5 represent the desired functional behavior of the system, while $RQ6$ and $RQ7$ are non-functional requirements. The overall goal – i.e., the union of all functional and non-functional requirements – may be accomplished in several ways, although there is some preferred (partial) ordering among the different actions that build the orchestration, e.g., the choice of the movie has to precede the choice of the theater. The UML Activity Diagram in Figure 1 (input activities in grey), models these precedence relationships.

To implement PD we may exploit the set of existing, publicly available Web services listed in Table I. In most cases there are different alternatives for the same service that are functionally equivalent, while in one case (the *searchRestaurant* action) we have two alternatives that differ in their parameters (i.e., coordinates or location name). Each service in the table may be associated to QoS data like response time (T) and reliability (R). These data may come from SLAs provided by service providers (as for the Azure Service Bus which reports in its SLA a reliability equal to 99.95%[1]), or they may come from direct measurements and estimates (e.g., [3], [11]). Next section recalls DSOL concepts and illustrates how we modified its architecture to support such QoS attributes.

TABLE I
PROGRAMMABLE DINNER SERVICES

Action	Return Value	Possible Providers
$getMap$	A map of a location	Google, Bing, MapQuest
$getDirections$	Directions to a destination	Google, MapQuest, Bing
$getForecasts$	Weather forecast	Wunderground, World Weather
$search$-$Restaurant$	List of restaurants given a location	Yahoo!, Google CityGridMedia
$search$-$Theater$	List of theaters given a location and a movie	Google, Yahoo!, CityGridMedia
$getReviews$	Restaurant reviews	Yelp, CityGridMedia
$getSimilar-Movies$	Similar Movies from a movie title	Rotten Tomatoes, TasteKid
$getMovie$-$Reviews$	Reviews of a list of movies	Rotten Tomatoes, NY Times Movie Reviews API
$sendMessage$	None	Nexmo, Hoiio

III. DSOL IN A NUTSHELL

In DSOL, a service orchestration is modeled declaratively by listing its *initial state*, its *goals*, and the set of *abstract* and *concrete actions* available in the domain of interest. At runtime, the DSOL Engine – *DEng* uses planning techniques to determine the actual flow of execution (i.e., which actions to execute and in which order) to achieve the orchestration's goals from the initial state. More specifically (see Figure 2) the DSOL infrastructure includes the elements described hereafter.

Abstract Actions. They are high-level descriptions of the primitive actions available in a given domain. DSOL relies on them at runtime as the building blocks to compose the orchestration. They are modeled in an easy-to-use, logic-like language, in terms of their *signature*, *precondition*, and *post-condition*. Listing 1 illustrates some abstract actions present in the ProgrammableDinner scenario. Notice how they closely reflect the activities described in the application requirements, leaving out all the implementation details, including the expected sequence of execution and the actual service binding.

```
action getForecast(Location)
pre: searchLocation(Location)
post: forecast(forecast_info, Location)

action searchRestaurants(Location)
pre: searchLocation(Location)
post: list_of(restaurants)

action getReviews(Places)
pre: list_of(Places)
post: reviewsIncludedTo(Places)

action createMapWithMarkers(Places)
pre: list_of(Places)
post: mapWithMarkers(Places)
```

Listing 1. Abstract actions

[1]http://www.windowsazure.com/en-us/support/sla/

Concrete Actions. They are the executable counterpart of abstract actions and model the concrete steps required to implement them, e.g., by invoking an external service or executing some code. They are modeled as Java methods, using the ad-hoc annotation *@Action* to refer to the abstract action they implement. In general, several concrete actions may be bound to the same abstract action. The DSOL Interpreter uses them to improve reliability: if the first bound concrete action fails, i.e., it returns an exception, it may try the alternative ones to successfully complete the given abstract action. Among concrete actions, DSOL distinguishes between *service* and *generic* actions. Service actions are abstract Java methods directly mapped to external services (e.g., see action *getForecastWithWunderground* in Listing 2). Service actions use the attribute *service* of the *@Action* annotation to reference the external service they are linked to. Using just a mnemonic label enables a loosely coupled model, where the details about the service (e.g., its URI and operation) is specified externally and can be changed at run-time.

```
@Action(service = "wunderground")
public abstract WundergroundResults
getForecastWithWunderground(String location);

@Action(service = "worldweatheronline")
public abstract WorldWeatherOnlineResults
getForecastWithWorldWeatherOnline(String location);

@Action(name = "getForecast")
@ReturnValue("forecast_info")
public List<Forecast>
WundergroundServiceWrapper(String location) {
 WundergroundResults results = getForecastWithWunderground(location);
 List<ForecastDay> result = results.getForecast().getSimpleforecast().getForecastday();
 List<Forecast> forecasts = new ArrayList<Forecast>();
 for (ForecastDay forecast : result)
  forecasts.add(new Forecast(forecast.getConditions()));
 return forecasts;
}

@Action(name = "getForecast")
@ReturnValue("forecast_info")
public List<Forecast>
WorldWeatherOnlineServiceWrapper(String location) {
 WorldWeatherOnlineResults results = getForecastWithWorldWeatherOnline(location);
 List<Weather> result = results .getData().getWeather();
 List<Forecast> forecasts = new ArrayList<Forecast>();
 for (Weather weather : result)
  forecasts.add(new Forecast(weather.getWeatherDesc().get(0).getValue()));
 return forecasts;
}
```

Listing 2. Concrete actions

Generic actions are ordinary Java methods to be used to perform general operations, like retrieving information from a database or pre/post-processing data between service invocations. As an example, actions *WundergroundServiceWrapper* and *WorldWeatherOnlineServiceWrapper* in Listing 2, represent two alternatives for the abstract action *getForecast*, each invoking a different external service (through concrete actions *getForecastWithWunderground* and *getForecastWithWorldWeatherOnline*) and adapting their results to a common interface, i.e., a list of *Forecast* objects, a concept that is part of the orchestration domain and not specific of any service.

Orchestration Interface, Initial State, and Goal. The *orchestration interface* formalizes how the orchestration is exposed as a web service, including the expected parameters and the returned object. The *initial state* models the set of facts that one can assume to be true at orchestration's invocation time. Finally, the *goal* is the set of facts that represent the expected state of the world after executing the orchestration. Listing 3 illustrates the initial state and part of the orchestration goal for the ProgrammableDinner example.

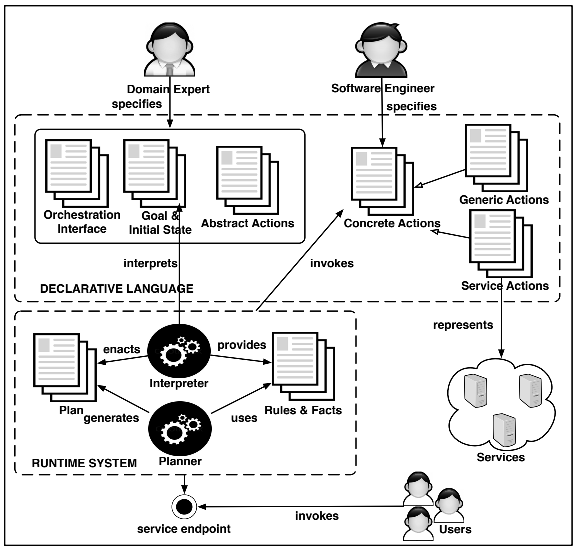


Fig. 2. The DSOL Architecture

```
start searchLocation(userDefinedLocation)
goal forecast(forecast_info, userDefinedLocation) and
     list_of(restaurants) and reviewsIncludedTo(restaurants) and
     mapWithMarkers(restaurants)
```

Listing 3. Initial state and goal

Execution. At the time a DSOL orchestration is invoked, its goal, initial state, and abstract actions are used by an internal *Planner* to build an abstract plan of execution. Listing 4 illustrates an execution plan for the goal expressed in Listing 3. It includes a list of abstract actions that leads the execution from the initial state to a state that satisfies the orchestration goal. In presence of multiple alternatives to reach the goal, the DSOL original Planner chooses one non-deterministically.

```
getForecast (userDefinedLocation)
searchRestaurants (userDefinedLocation)
addReviewsTo (restaurants)
createMapWithMarkers (restaurants)
```

Listing 4. A possible execution plan

This plan is taken and enacted by associating each step (i.e., each *abstract action*) with a *concrete action* (chosen non-deterministically among those available), which is executed by the *Interpreter*, possibly invoking external services. To properly execute the plan, the Interpreter keeps a map, called *Instance Session*, between the abstract objects part of the plan and the actual Java objects processed at runtime. The Instance Session is initialized with the arguments sent by the client while invoking the composite service and it can be changed while the plan is executed (e.g., see the *@ReturnValue* annotation in Listing 2). It is also important to note that while the plan is described as a sequence of actions, the Interpreter parallelizes execution as much as possible, by invoking each action as soon as its precondition is satisfied.

Re-Binding and Re-Planning. If something goes wrong during execution (e.g., an external service is faulty or unavailable), first a different concrete action for the failing abstract action is tried (if any), then, if this does not allow to get around the problem, the Planner is invoked again to find a different course of actions that could skip the failed step. By comparing the old and the new plan, considering the current state of execution, the system computes the set of actions that need to be *compensated* as they have already been executed but are not part of the new plan. Ad-hoc *compensation actions* are defined in DSOL as special concrete actions.

IV. Dynamic Service Orchestration with Q-DSOL

A. Adding QoS to DSOL

QoS-aware orchestration infrastructures must consider non-functional aspects in: (1) choosing the best services to be included in the orchestration, and (2) designing the orchestration structure. To achieve this objective, we had to change both DSOL language and run-time system. Concerning the former, Q-DSOL adds QoS profiles to concrete actions using the *@QoSProfile* annotation, while it extends the goal definition to include the non-functional requirements of the orchestration. For example, Listing 5 shows the first concrete action of Listing 4 augmented with information about its expected reliability and response time. These data are provided by the orchestration designer and, as previously mentioned, they may be obtained by direct measurements or SLAs.

```
@Action(service = "wunderground")
@QoSProfile(metrics={"reliability", "response_time"}, values={0.995,300})
public abstract WundergroundResults
getWeatherForecastWithWunderground(String location);
```

Listing 5. Concrete actions with QoS Profile

Similarly, Listing 6 illustrates the goal definition corresponding to requirements $R6$ and $R7$. Notice that to cope with conflicting QoS requirements, we may specify both desired bounds and preferred optimizations. Depending on the QoS metric, the desired bounds represent the lower (e.g., for reliability) or upper (e.g., for time) bounds that the orchestration must meet. In our example, according to $R6$, we want a reliability of at least 0.99 and a response time of at most $1.5s$. Since different configurations could meet these bounds, we may also ask to optimize against a specific metric by using the *min* and *max* keywords. In our example, according to $R7$, we ask to minimize the response time.

```
goal RQ1 and RQ2...RQ5
and desired(reliability , 0.99) and desired(response_time , 1500)
and min(response_time)
```

Listing 6. Goal Definition Including QoS Requirements

These changes in the modeling language reflect to changes in the run-time system. Indeed, the original DSOL Planner (see Section III) generates all possible plans that satisfy the functional requirements of the orchestration, while the Interpreter chooses non-deterministically one of them and executes it, binding each abstract action to one among the available concrete actions. Q-DSOL has to change this behaviour to consider QoS, since (1) the plan to be executed and (2) its bindings to concrete actions are relevant for QoS. In particular, Q-DSOL formalizes the problem of finding the plan to run and the concrete actions to bind as an *optimization problem*, and it exploits *linear programming techniques* [10] to solve it.

The optimization problem includes:

- A set of abstract actions $A = \{a_1, \ldots, a_n\}$ where $n \geq 1$.
- For each abstract action a_i, a set of concrete actions $C_i = \{c_{i,1}, \ldots, c_{i,m_i}\}$ where $m_i \geq 1$. Each concrete action $c_{i,j}$ is characterized by a response time $t_{i,j} > 0$ and a reliability $r_{i,j} \in [0, 1]$.
- A set of plans $P = \{P_1, \ldots, P_l\}$ where $l \geq 1$, each modeled as a set of abstract actions.

We define a *binding variable* $s_{i,j,x} \in \{0, 1\}$ (where $i \in [1, n]$, $j \in [1, m_i]$, and $x \in [1, l]$) to indicate if the concrete action $c_{i,j}$ is bound to the abstract action a_i in plan P_x (i.e., $s_{i,j,x} = 1$) or vice versa (i.e., $s_{i,j,x} = 0$). This assignment must meet the following constraints:

$$(\forall x, i | a_i \in P_x) \sum_{0<j\leq m_i} s_{i,j,x} \leq 1 \quad (1)$$

$$(\forall x, i | a_i \notin P_x) \sum_{0<j\leq m_i} s_{i,j,x} = 0 \quad (2)$$

$$(\forall x, i, j | s_{i,j,x} \neq 0) \rightarrow (\forall x', i', j' | x' \neq x)\ s_{i',j',x'} = 0 \quad (3)$$

Equation 1 and 2 together indicate that we bind at most one concrete action to every abstract action part of a plan, while we leave unbound those abstract actions that are not part of a plan. Equation 3 indicates that if we bind a concrete action in a plan x, we cannot bind any action in plans different from x. In other words we bind one and only one plan. Accordingly, a valid assignment to binding variables $s_{i,j,x}$ returns a single plan bound to a set of concrete actions.

Given these definitions, our optimization problem boils down to find the *optimal assignment* to $s_{i,j,x}$, i.e., the assignment corresponding to a bound plan that satisfies QoS requirements. This can be formalized by introducing two aggregation functions f_R and f_T, for reliability and response time, respectively[2]. In particular, f_T is the sum of all $t_{i,j}$ of each concrete action such that $s_{i,j,x}$ is set to one[3]:

$$f_T = \sum_{\forall i,j,x} s_{i,j,x} \times t_{i,j}$$

Similarly, f_R aggregates reliability by multiplying $r_{i,j}$ to the power of $s_{i,j,x}$:

$$f_R = \prod_{\forall i,j,x} r_{i,j}^{s_{i,j,x}}$$

Given these two aggregation functions and indicating with G the set of all possible assignments to binding variables $s_{i,j,x}$, we may define the optimization problem looking at the goal definition. For example, recalling Listing 6, we have that the optimal assignment is defined as follows:

$$\begin{aligned} \underset{\forall g \in G}{\text{minimize:}} \quad & f_T, \\ \text{subject to:} \quad & f_T < 1500ms \\ & f_R > 0.99 \end{aligned}$$

[2]In presence of other, user defined metrics, we add similar functions.

[3]For parallel actions, Q-DSOL considers only the slowest one.

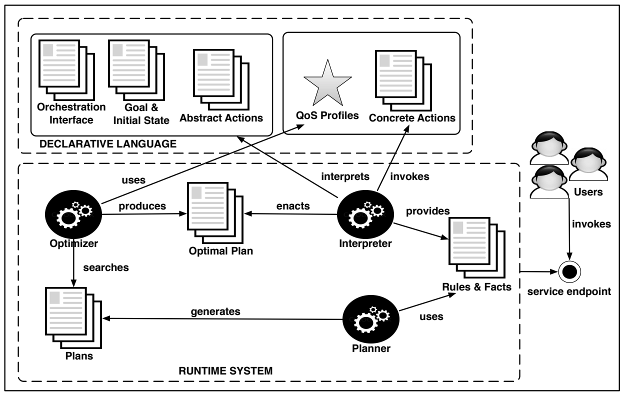


Fig. 3. The QoS-Aware DSOL approach

Since f_R is non linear with respect to the optimization variables $s_{i,j,x}$, we linearize it using a typical technique [10], which applies the logarithm to both sides of the equation: $\log(f_R) > \log(0.99)$.

The optimization problem above is solved by the *Optimizer*, an ad-hoc component of the run-time system of Q-DSOL, which internally relies on AMPL [12] to find the optimal assignment to $s_{i,j,x}$, i.e., to choose the plan to execute and how to bind concrete actions to abstract ones. The architecture of Q-DSOL is reported in Figure 3. At run-time, Q-DSOL applies two additional mechanisms: (1) *Adaptive Re-Binding* and (2) *Adaptive Re-Planning*, to further optimize the QoS perceived by the end user based on the actual situations encountered. Next paragraphs illustrate these two mechanisms referring to the PD example.

B. Maximizing Performance via Adaptive Re-binding

If the orchestration goals specify that the response time must be minimized, the Q-DSOL engine does not limit to blindly execute the plan returned by the Optimizer, but it applies an *Adaptive Re-Binding* strategy. With this term we indicate the fact that it evaluates, at every step of execution, alternative bindings to abstract actions that could decrease the expected response time of the orchestration. If a better alternative (i.e., a faster concrete action) is found, the engine re-binds the current action automatically.

Notice that better alternative actions may be actually found since, even if the condition of optimality concerning the bindings produced by the Optimizer holds before starting the execution of the plan, it may not hold anymore during its execution. Indeed, during execution, the QoS values associated to concrete actions can be updated. In particular, the reliability of actions already performed can be set to one[4]. Increasing the reliability of certain actions implies that other concrete actions, initially discarded by the Optimizer because too unreliable to reach the specified bound, may now become eligible for execution, and they could be actually chosen if lead to a lower response time. In other words, their high probability of failure

[4]If a concrete action appears more then once in a plan, we set reliability equal to one only for the invocations already occurred.

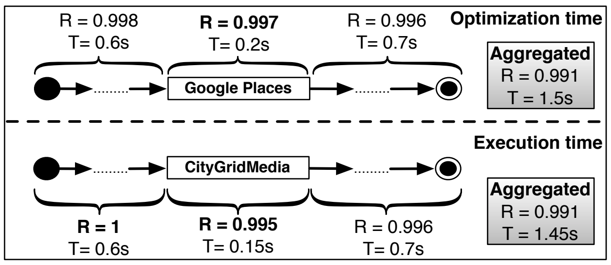


Fig. 4. Adaptive Re-Binding Example

was compensated by the already successfully run actions, and they can now be taken into consideration.

Let us illustrates this scenario by recalling our ProgrammableDinner example. Let us imagine that the Q-DSOL engine has already executed part of the plan: the next action to execute is the *searchTheater* operation. As reported in Table I we have three alternatives to implement this step and let us assume Google was the concrete action chosen by the Optimizer. We can decompose the overall reliability of the plan in three contributes. The first regards actions already performed, the second is the reliability of the concrete action currently bound to searchTheater (i.e., Google), and the third is the expected reliability of concrete actions to be executed after searchTheater. If the first contribute was equal to 0.998 at optimization time (see Figure 4), now (i.e., at run-time) we may assume it being one. This enables the CityMediaGrid concrete action: it has a lower reliability than Google, but now that we updated the reliability of the preceding actions it is reliable enough to reach $RQ6$. Being faster than Google it becomes the best choice to minimize the response time, i.e., to satisfy $RQ7$.

This kind of reasoning is performed by Q-DSOL at every step of the plan. Indeed, this is a fast search (we proceed incrementally, focusing only on the next abstract action to execute) that may considerably improve the overall response time of the orchestration, especially in presence of very efficient but unreliable services.

C. Maximizing Reliability in Presence of Failures

If we focus on reliability, we may observe that in defining our optimization problem we did not take into consideration the DSOL ability to adapt the orchestration at run-time, re-binding faulty actions to alternative services and, if this was not enough, re-building the entire plan to circumvent multiple failures. This choice is motivated by the impossibility to correctly estimate and account how this re-binding and re-planning mechanisms (to improve reliability) may impact response time[5]. This means that the optimal plan initially found by the Optimizer, including the mapping to concrete actions, is expected to provide the desired reliability by itself. In our ProgrammableDinner example, the optimal plan coming from the Optimizer will succeed 99% of the times ($RQ6$).

[5]Indeed, re-planning and re-binding may require compensating one or more actions, which further impact execution time.

On the other hand, the adaptive features of DSOL are there, and they can be used to improve the reliability perceived by the end-user. In practice, in presence of a faulty service, Q-DSOL re-binds the corresponding abstract action to an alternative concrete action. This way, an invocation that should fail can be saved and terminate correctly, contributing to increase the overall reliability.

Moreover, if all alternatives fails, Q-DSOL builds an alternative plan that skips the failed action (eventually compensating already performed actions that are not part of the new plan), thus enabling new opportunities to end the orchestration correctly. In this re-planning case, the Optimizer is invoked again to choose the new optimal plan among the alternative ones.

It is important to notice that in all these cases response time requirements are not guaranteed anymore. The engine is doing its best to overcome a failure and the completion of the orchestration is the current priority. On the other hand, we do not ignore response time. At re-binding time Q-DSOL chooses alternative actions ordered by response time, while at re-planning time, it runs the Optimizer to choose the best plan also considering response time.

Finally, it is important to notice that Q-DSOL updates automatically the QoS attributes of concrete actions based on their observed behaviour. This implies that two subsequent invocations to the same orchestration may be served by different services or different plans if the conditions change between the first and the second invocation.

D. Domain Specific QoS Metrics

So far we only considered reliability and response time as QoS metrics. However, orchestration designers may need to consider other QoS aspects, such as costs or availability, to model domain specific requirements. To cover these scenarios, Q-DSOL allows domain specific QoS metrics to be easily defined.

Domain Specific QoS Metrics (DSQM) are characterized by their name (e.g., *"cost"* or *"availability"*) and by two *aggregation operators*: $\langle s, p\rangle$, which are used by Q-DSOL to calculate the DSQM value for an entire plan starting from the DSQM value of each action. In particular, both s and p can be algebraic operators ($+, -, \times, /$) or simple functions (*min*, *max*, *abs*, *avg*). Operator s indicates to Q-DSOL how to aggregate the value of actions executed in sequence, while operator p indicates how to aggregate parallel actions. As an example, $\langle +, max\rangle$ are the operators for the pre-defined response time metric, while $\langle \times, \times\rangle$ are those for reliability. The definition of DSQM is characterized also by the *metric limit* which may be *upper* or *lower*. The former indicates that the optimal plan should have *at least* the desired value expressed by the goal definition, vice-versa the latter indicates that the desired value will be considered as a maximum limit for the DSQM.

The introduction of DSQM affects both Q-DSOL language and run-time system. Let us start from the language with an example that extends the ProgrammableDinner orchestration adding the following requirement:

- $RQ8$: The total cost for executing the orchestration should not exceed 3$.

Imagine that both the *sendMessage* and *getForecast* services charge a small fee for each invocation, the exact amount depending from the service provider. To satisfy $RQ8$ we define the DSQM *"cost"* (C) as reported in Listing 7. It uses the sum as aggregate operator both for sequential and parallel actions.

```
define(cost, aggregation<+, +>, limit<upper>)
goal RQ1 and RQ2...RQ5
and desired(reliability, 0.999) and desired(response_time, 1500)
and desired(cost, 3) and min(response_time)
```

Listing 7. DSQM Definition

Given a DSQM, at run-time Q-DSOL behaves as explained in the previous sections except for the optimization problem, which now includes additional aggregation functions, similar to f_R and f_T and defined using the respective aggregation operators. For the cost example, this means adding the aggregation function f_C, defined using the sum to aggregate both sequential and parallel actions. We use it, together with the definition of the metric limit, to define an additional constraint $f_C < 3$ starting from the goal in Listing 7.

V. Q-DSOL at Work

This section discusses the validation of Q-DSOL with a two step approach. First, it evaluates the overhead of Q-DSOL[6]. Secondly, it investigates the potential speed-up it provides. All the data illustrated in this section has been obtained using our Q-DSOL implementation, an open source tool publicly available at *http://www.dsol-lang.net/*. Due to the lack of space we cannot provide the full set of experiments, but we only report the most significant results.

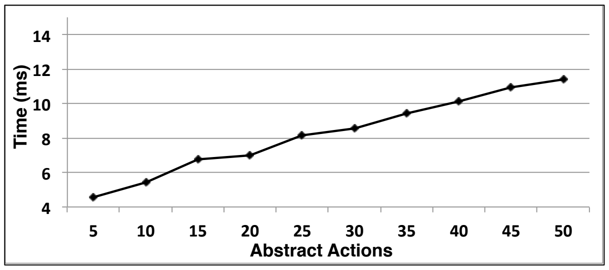
Our experiments were carried out in an local server configured to emulate a typical application server used to deploy service-based applications. Such server had the following configuration: Intel Core i5 processor, 4GB RAM and Ubuntu Linux (version 11.10) operating system.

Q-DSOL Overhead: Determining the optimal plan given the set of available actions and the orchestration goal, introduces an overhead at execution time, which we measured as follows. First of all we measured the overhead considering plans of variable dimensions in terms of the number of abstract actions considered. Figure 5(a) reports the optimization time for plans composed by an increasing number of abstract actions. In this experiment, each abstract action has three alternative concrete actions. The measured optimization time account for less than $12ms$ for a plan comprising fifty abstract actions, which is a perfectly acceptable overhead considering that the resulting orchestration would probably involve a number of service invocations close to fifty[7], which needs seconds to execute.

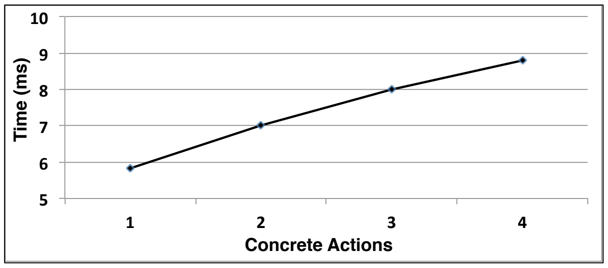
The second experiment we made kept constant the number of abstract actions (i.e., the plan length) increasing the number of concrete alternatives associated to each abstract actions.

[6]This overhead does not include the time required to find the possible plans, as such time is strongly related to the abstract actions domain. For further details we refer to [9].

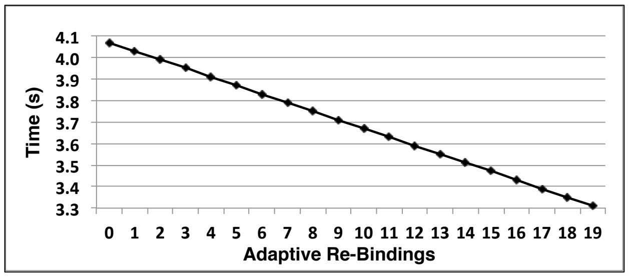
[7]Most Q-DSOL actions results in invoking an external service.



(a) Optimization Time over Abstract Actions.



(b) Optimization Time over Concrete Actions



(c) Execution Time over Re-Bindings.

Fig. 5. Q-DSOL Validation

Figure 5(b) reports the results obtained with a plan composed of twenty abstract actions. Even in this case the optimization time is negligible w.r.t. the plan execution time: less than $9ms$ in the worst case of four concrete alternatives for each abstract action. Note that, in practice, the number of alternative concrete actions that can be found for each abstract action is typically small, for example, in our ProgrammableDinner orchestration we was not able to find more than three alternatives for each action.

Q-DSOL Speed-Up: Once we verified that our general optimization strategy introduces a negligible overhead w.r.t. a non optimized solution, we were interested in measuring the impact on performance of our mechanisms.

We started measuring the speed-up gained through the adaptive re-binding mechanism described in Section IV-B. We considered a plan composed by twenty abstract actions with two concrete actions each. The first is selected at optimization time while the second becomes the more convenient option at execution time because it has a response time 20% faster and its lower reliability is compensated by the already executed actions. In absence of re-binding we measured an average execution time for the orchestration of $4.069s$. Then we activated the re-binding mechanism and let it run a growing number of times, from one to 19 (we have a total of 20 actions). The results we measured are reported in Figure 5(c). We notice that we gain a linear speedup which is maximum when we let Q-DSOL re-bind every possible action. The speedup we measure in this case is 18.62%. This is very close to 20%, which is the maximum theoretical speedup we could obtain under this scenario. This demonstrates that the advantages of the re-binding mechanism come at a negligible cost: the difference between 18.62% and 20%, which is the overhead of finding the best alternative and re-binding it.

The last test we performed was to measure the overall speed-up of Q-DSOL w.r.t. a non optimizing solution. To do so, we took our ProgrammableDinner example and measured the actual (average) execution time of all the alternative services that can be used to compose the orchestration. In absence of an optimizer like the one integrated in Q-DSOL, we could expect that a standard engine (like DSOL) takes a random plan, so we measured the average execution time of all possible plans: it takes $2s$ to complete. Than we let Q-DSOL run the orchestration asking it to minimize the execution time. The result we measured was $1.2s$, which corresponds to a speed-up of 40%.

In general, those results demonstrate that the optimization problem solved by Q-DSOL and the adaptive re-binding mechanism it implements, introduce a limited and negligible overhead with respect to the execution of the entire orchestration, while providing a significant potential speed-up in terms of response time (not to mention the advantages in terms of reliability and the fact that it provides a guaranteed QoS as specified by the user in the goal).

VI. Related Work

DSOL was designed as an alternative to traditional orchestration languages such as BPEL, BPMN, Jolie [17] and JOpera [20]. Indeed DSOL has been designed as a declarative approach to support the definition of flexible and self-adaptive service orchestrations able to cope with unexpected behaviors at run- time. We refer to [9] for a complete comparison among DSOL w.r.t. to existing orchestration languages and, hereafter, we focus on approaches related to QoS.

Many existing approaches have an explicit support for QoS with different levels of abstractions and leveraging on a plethora of different techniques. For example, Menascé [16] discusses QoS in the domain of services, introducing the response times, availability, security, and throughput as QoS parameters. His paper also discusses the need of SLAs without advocating any specific model to manage, aggregate and optimize QoS behaviors of service orchestrations.

In particular, concerning optimization, many approaches exploit linear programming to manage QoS for orchestrations. For example, Aggrawal et al. [1] view QoS-based composition as a constraint satisfaction/optimization problem and find an optimal solution by applying integer linear programming. Zeng et al. [22] present comprehensive research about QoS modeling and QoS-aware compositions. In particular, they use statecharts to model orchestrations in which services are selected from a pool of alternative services using linear programming techniques such that it optimizes a local as well as global QoS criteria. Alternatively to linear programming, in [18], the authors leverage on fuzzy distributed constraint satisfaction techniques for finding the optimal orchestration. All these approaches differs from our proposal in many aspects. First we do not consider only alternative bindings in finding the optimal orchestration but we also consider structural alternatives (i.e. plans) in finding the optimal solution. Secondly, we support domain specific metrics and adaptivity in terms of adaptive re-binding and re-planning.

Concerning the aggregation functions for QoS metrics, other existing approaches propose similar techniques aimed at aggregating metrics (e.g., [4], [14], [21]). For example, Cardoso et al. [21] compute aggregate QoS by applying a set of reduction rules to the workflow until one atomic task is obtained. In addition, other approaches support custom specific metrics (i.e., DSQM) such as [4], [15]. However, all these approaches may not guarantee the optimal solution with respect to QoS even if they may be suitable where optimality is not mandatory and execution efficiency is preferred. Among these works let us mention the approaches based on genetic programming such as [5], [7] or on heuristics (e.g., [2]).

Finally, concerning specifically adaptivity and re-planning we may mention respectively [8], [13] and [6]. The first two approaches do not focus on QoS, conversely the third one provides an efficient re-planning technique that, however, do not guarantees the optimal solution. Summing up, none of the existing approaches, at the best of our knowledge, mix together an optimal solution, custom specific metrics and adaptive capabilities as Q-DSOL with the adaptivity techniques of re-binding and re-planning. In addition, this is the first approach that combines planning together with optimization which allow the easy of use of declarative languages with the guarantee of optimality.

VII. Conclusions and Future Work

In this paper we extended our previous work, DSOL, to support quality requirements and adaptivity. We introduced an optimizer which generates an optimal solution in terms of execution plan and bindings. We discussed all these features relying on a service orchestration built out of real publicly available services. Furthermore, we measured the overhead of the approach and the potential speed-up to demonstrate the pro and cons of our solution. The contribution of the paper is twofold. First of all, to the best of our knowledge this is the first approach that relies on planning techniques together with linear optimization integrated by a unifying declarative language. Secondly, we provided a novel solution that further optimizes response time and reliability via adaptive re-binding and re-planning. Q-DSOL is part of a long running research stream and its future work includes building a IDE, possibly integrated in a widely adopted tool such as Eclipse, to further simplify the definition of abstract actions, goals, and orchestration interfaces. As for the Q-DSOL run-time system, while the current prototype is operational–and downloadable–there is still space to further improve performance and robustness.

Acknowledgment

This research has been funded by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom and Programme FP7-PEOPLE-2011-IEF, Project 302648-RunMore.

References

[1] R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint driven web service composition in meteor-s. In *Services Computing, 2004.* IEEE.

[2] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz. Heuristics for qos-aware web service composition. In *Web Services, 2006. ICWS'06. International Conference on*, pages 72–82. IEEE, 2006.

[3] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *Software Engineering, IEEE Transactions on*, 37(3):387 –409, may-june 2011.

[4] G. Canfora, M. Di Penta, R. Esposito, F. Perfetto, and M. Villani. Service composition (re) binding driven by application–specific qos. *Service-Oriented Computing–ICSOC 2006*, pages 141–152, 2006.

[5] G. Canfora, M. Di Penta, R. Esposito, and M. Villani. An approach for qos-aware service composition based on genetic algorithms. In *GECCO 2005.* ACM.

[6] G. Canfora, M. Di Penta, R. Esposito, and M. Villani. Qos-aware replanning of composite web services. In *ICWS 2005.* IEEE.

[7] G. Canfora, M. Di Penta, R. Esposito, and M. Villani. A framework for qos-aware binding and re-binding of composite web services. *Journal of Systems and Software*, 81(10):1754–1769, 2008.

[8] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M. Shan. Adaptive and dynamic service composition in eflow. In *Advanced Information Systems Engineering*, pages 13–31. Springer, 2000.

[9] G. Cugola, C. Ghezzi, and L. S. Pinto. DSOL: a declarative approach to self-adaptive service orchestrations. *Computing*, pages 1–39. 10.1007/s00607-012-0194-z.

[10] G. Dantzig. *Linear programming and extensions.* Princeton Univ Pr.

[11] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Model evolution by run-time parameter adaptation. In *ICSE '09*, pages 111–121, 2009.

[12] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A mathematical programming language.* AT&T Bell Laboratories, 1987.

[13] D. Ivanovic, M. Carro, and M. Hermenegildo. Towards data-aware qos-driven adaptation for service orchestrations. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 107–114. IEEE, 2010.

[14] M. Jaeger, G. Rojec-Goldmann, and G. Muhl. Qos aggregation in web service compositions. In *e-Technology, e-Commerce and e-Service. EEE'05. Proceedings. The 2005 IEEE International Conference on.*

[15] Y. Liu, A. H. Ngu, and L. Z. Zeng. Qos computation and policing in dynamic web service selection. WWW Alt. '04. ACM, 2004.

[16] D. Menasce. Qos issues in web services. *Internet Computing, IEEE*, 6(6):72 – 75, nov/dec 2002.

[17] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. Jolie: a java orchestration language interpreter engine. *Notes Theor. Comput. Sci.*

[18] X. T. Nguyen, R. Kowalczyk, and M. T. Phan. Modelling and solving qos composition problem using fuzzy discsp. In *Web Services, 2006. ICWS '06. International Conference on*, pages 55 –62, sept. 2006.

[19] M. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Communications of the ACM*, 46(10):25–28, 2003.

[20] C. Pautasso and G. Alonso. Jopera: A toolkit for efficient visual composition of web services. *Int. J. Electron. Commerce*, 9, 2005.

[21] A. Sheth, J. Cardoso, J. Miller, K. Kochut, and M. Kang. Qos for service-oriented middleware. In *Proceedings of the Conference on Systemics, Cybernetics and Informatics*, pages 130–141, 2002.

[22] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *Software Engineering, IEEE Transactions on*, 30(5):311–327, 2004.