Towards Self-adaptation for Dependable Service-Oriented Systems

Valeria Cardellini[1], Emiliano Casalicchio[1], Vincenzo Grassi[1], Francesco Lo Presti[1], and Raffaela Mirandola[2]

[1] Università di Roma "Tor Vergata", Viale del Politecnico 1, 00133 Roma, Italy
{cardellini,casalicchio}@ing.uniroma2.it,
{vgrassi,lopresti}@info.uniroma2.it
[2] Politecnico di Milano, Piazza Leonardo Da Vinci 32, 20133 Milano, Italy
mirandola@elet.polimi.it

Abstract. Increasingly complex information systems operating in dynamic environments ask for management policies able to deal intelligently and autonomously with problems and tasks. An attempt to deal with these aspects can be found in the Service-Oriented Architecture (SOA) paradigm that foresees the creation of business applications from independently developed services, where services and applications build up complex dependencies. Therefore the dependability of SOA systems strongly depends on their ability to self-manage and adapt themselves to cope with changes in the operating conditions and to meet the required dependability with a minimum of resources. In this paper we propose a model-based approach to the realization of self-adaptable SOA systems, aimed at the fulfillment of dependability requirements. Specifically, we provide a methodology driving the system adaptation and we discuss the architectural issues related to its implementation. To bring this approach to fruition, we developed a prototype tool and we show the results that can be achieved with a simple example.

1 Introduction

The SOA paradigm emphasizes the construction of software systems through the dynamic composition of network-accessible services offered by loosely coupled independent providers. As a consequence, such systems have to tackle problems caused by component services becoming unreachable because of connection problems, or changing their delivered Quality of Service (QoS), or even being turned off. These problems have a direct impact on the system dependability, both in terms of its *availability* (ability to accept a service request, when a service offered by the system is invoked) and of its *reliability* (ability to successfully complete the requested service, once a request has been accepted) [1]. Thus, guaranteeing a high dependability level for SOA systems is a key factor for their success in the envisioned "service market", where service providers compete by offering services with different quality and cost attributes [2,3].

R. de Lemos et al. (Eds.): Architecting Dependable Systems VI, LNCS 5835, pp. 24–48, 2009.

Achieving this goal is a challenging task, as the system must face the high variability of the execution environment, the dependability requirements of different classes of users, and the limits on the available resources, needed to keep the system cost within a given budget.

A promising way to cope with these problems is to make the system able to self-adapt to changes in its environment (available resources, type and amount of user demand), by autonomously modifying at runtime its behavior or structure. In this way, the system can timely react to (or even anticipate) environment changes, trying to use at best the available resources, thus avoiding long service disruptions due to off-line repairs [4,5].

Some general proposals about how to architect a self-adaptable software system have already appeared [6,7,8,9]. These proposals suggest architectural frameworks which can be used to support the implementation of suitable adaptation methodologies, possibly tailored to specific application domains.

In this respect, methodologies that can be implemented within these architectural frameworks to drive the adaptation of a SOA system have been already presented. Some of them specifically focus on the fulfillment of dependability requirements (e.g., [10]), while others consider multiple quality attributes including dependability (see, for example, [11]). Most of these methodologies consider exclusively a single kind of adaptation mechanism, based on *service selection*. According to this mechanism, the set of component services used to build a composite SOA system is dynamically selected and bound to the system, based on the current operating environment conditions and system requirements. The methodology presented in [10] considers instead a different kind of adaptation mechanism, based on *architecture selection*. In this case, it is the (redundancy based) architecture for the service composition which is dynamically selected, to maintain the system ability to meet a dependability requirement.

The scenario these methodologies focus on generally consists of a single request addressed to a composite SOA system, considered independently of other requests which could be addressed to the same system. The aim is to determine the adaptation action which is (possibly) optimal for that single request, considering a given set of quality requirements and the current conditions of the operating environment. A limit of these methodologies is that they consider a single type of adaptation mechanism (either service selection or architecture selection) when they try to determine the best possible adaptation action. Instead, considering simultaneously a broader range of adaptation mechanisms could increase the system flexibility in adapting to different environments and requirements.

Moreover, we point out that methodologies aimed at determining adaptation actions for single service requests, independently of other concurrent requests, could incur in problems under a sustained traffic of requests addressed to a composite SOA system. Indeed, the "local" adaptation action they determine could conflict with adaptation actions determined for other concurrent requests, leading to instability and management problems.

Another potential limitation of these methodologies is that they generally formulate the problem to be solved as a NP-hard problem, which could thus result too complex for runtime decisions. This aspect is particularly critical in a SOA environment, where adaptation actions are likely to be calculated relatively often, due to its highly dynamic nature.

Based on these considerations, the main goal of our proposal is to provide ideas towards the realization of an adaptable SOA system that can flexibly base its adaptation actions on both kinds of adaptation mechanisms outlined above, to meet its dependability objectives. Besides providing a suitable modeling methodology for this purpose, we also suggest a possible architectural framework for its implementation. This architecture can be seen as an instantiation for the SOA domain of the general architectural frameworks outlined above, with a focus on the fulfillment of dependability requirements.

Moreover, differently from other approaches, we assume an operating scenario where a quite sustained traffic of requests is addressed to a SOA system. Hence, rather than trying to determine adaptation actions for each single request, our approach is aimed at determining adaptation actions for *flows* of requests. A potential drawback of our approach is that we loose the possibility of customizing the adaptation action for each request. However, in the scenario we consider, performing a *per-request* rather than a *per-flow* adaptation could cause an excessive computational burden. For example, the Amazon e-commerce platform, described in [12], comprises hundreds of services and tens of millions requests, which make the per-request approach hardly feasible. In addition, our per-flow approach allows us to deal simultaneously with different flows of requests, each with possibly different dependability requirements, thus possibly allowing a better balancing among different flows in the use of the available third-party services.

We present our approach from the perspective of a composite SOA system provided by an intermediary broker. The broker composes, according to some business logic, functionalities implemented by third-party services to offer a new added-value service. In doing this, it wants to guarantee to its users a given dependability level, maximizing at the same time an utility function (e.g. its income).

To achieve these goals within a changing environment (as it is typically the case for SOA systems), the broker adapts the system it manages in response to detected events. To this end, the broker maintains a model of the composite service it offers and of its environment, keeping it up to date thanks to a continuous monitoring activity. This model is used to determine adaptation actions in response to detected changes.

The events that may trigger an adaptation include both "normal" events like the arrival or departure of a user (with the related dependability requirements), and "abnormal" events like the unreachability of a component service or a relevant change of its QoS attributes. The adaptation actions performed by the broker are based on both the service selection and architecture selection mechanisms. Indeed, the broker can adapt the composite service it offers by

redefining the binding between incoming requests and component services (service selection), and by restructuring the composite service architecture (architecture selection).

We formulate the problem of determining the adaptation action triggered by some event as a Linear Programming (LP) optimization problem, which can be efficiently solved via standard techniques, and is therefore suitable for making runtime decisions. We have presented a preliminary version of this LP-based approach in [13]. However, in that paper we only considered service selection as the adaptation mechanism, while the problem formulation proposed in this paper considers also the modification of the service architecture.

The paper is organized as follows. In Sect. 2 we present a possible architecture of a self-adaptable SOA system that implements our proposed adaptation methodology. In Sect. 4 we present a mathematical formulation of the system model used to determine the adaptation actions, and discuss how to calculate the value of the dependability and cost attributes used in this model. In Sect. 5 we present the results of some numerical experiments. In Sect. 6 we discuss related work. Finally, we draw some conclusions and give hints for future work in Sect. 7.

2 General Architecture

In this section, we define the composite service model we refer to, and the type of contract used for the specification of the respective obligations and expectations of the service users and providers. Then, we outline the architecture of the broker that manages the composite service and its adaptation.

2.1 Composite Service

The system managed by the broker consists of a composite service, i.e., a composition of multiple services in one logical unit in order to accomplish a complex task. The composition logic can be abstractly defined as an instance of the following grammar:

$$S ::= S_1|S_2|...|Sn|seq(S+)|loop(S)|sel(S+)|par\_and(S+)|par\_or(S+)$$

where S_i denotes a single service, while $S+$ denotes a set of one or more services. Hence, a composite service can be either a single service, or the composition of other services according to the composition rules: $seq, loop, sel, par\_and, par\_or$. Table 1 summarizes the intended meaning of these rules. We point out that the above grammar is purposely abstract, as it intends to specify only the structure of a composite service. Details such as how to express the terminating condition for a loop are therefore omitted. The grammar does not capture all the possible composition rules (a broader set of composition rules is presented, for example, in [14]), but includes a significant subset. Table 1 also shows the mapping between these rules and the constructs of two well known service workflows specification languages: BPEL [15] and OWL-S [16]. For BPEL, the mapping refers

Table 1. Meaning of the grammar rules and mapping with the constructs of BPEL and OWL-S

Rule	Meaning	BPEL	OWL-S
seq	sequential execution of activities	`sequence`	`Sequence`
loop	repeated execution of activities in a loop	`while`, `repeatUntil`, `forEach`	`Repeat-While`, `Repeat-Until`
sel	conditional selection of activities	`if-elseif-else` `switch`	`If-Then-Else`
par_and	concurrent execution of activities (with complete synchronization)	`flow`	`Split-Join`
par_or	concurrent execution of activities (with 1 out of n synchronization)	`pick, forEach`	`Choice`

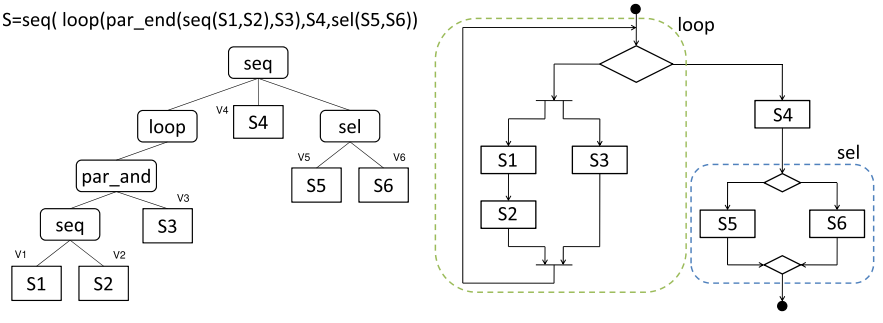


Fig. 1. An example of composite service instance derived form the proposed grammar, and its graphical representations: the syntax tree (left); the activity diagram (right)

to the structured style of composition (rather than to its graph-based one, using control links). Figure 1 shows a composite service instance that can be derived from this grammar, and the corresponding graphical representation in form of a syntax tree and activity diagram. From a semantic viewpoint, the instance shown in Fig. 1 abstractly represents the business logic of a composite service, where each S_i denotes a functionality (*abstract service*) needed to carry out its overall task. Each abstract service must then be bound to a *concrete service* that actually implements it. The overall dependability and cost of the composite service thus depend on the dependability and cost of the concrete services bound to its abstract services. In our approach, we assume that the involved parties state the required values for these attributes in a contract, whose schema is outlined in the next subsection.

2.2 Contract Definition

As usual in the SOA environment, we assume that the interactions between service requesters and service providers are regulated by a Service Level Agreement

(SLA), i.e., a contract which explicitly states the respective obligations and expectations [17]. This contract specifies the conditions for service delivery, including the quality and quantity levels (e.g., the load that the user can charge) of the provided service, its cost, duration, and penalties for non-compliance.

According to what discussed in the introduction, we consider in our approach SLAs stating conditions that should hold globally for a *flow* of requests generated by a user.

In general, a SLA may include a large set of parameters, referring to different kinds of functional and non-functional attributes of the service, and different ways of measuring them (e.g., averaged over some time interval) [18,17]. In this paper, we restrict our attention to the *average value* of the dependability attribute, globally experienced by all the requests belonging to the flow generated by a user. Hence, the SLA model we consider includes the following quantities:

- a_{min}: a lower bound on the service average dependability expected by a service user;
- L: an upper bound on the load the user is allowed to submit to the service, expressed in terms of average rate of service invocations (invocations/time unit);
- c: the unitary service cost paid by the user for each submitted request.

The broker that manages the composite service acts as an intermediary between the users of the composite service and the providers of the used component services, performing a role of service provider towards its users, and being in turn a user for the providers of the concrete services it uses to implement the composite service itself. Hence, it is involved in two types of SLAs, corresponding to these two roles: we call them *SLA-P* (provider role) and *SLA-R* (requester role). Both these SLAs are defined according to the triple described above, *i.e.*, $SLA\text{-}P = SLA\text{-}R = \langle a_{min}, L, c \rangle$.

In the case of the SLAs-P between the composite service users and the broker, we assume in our approach that the value of their parameters is the result of an individual negotiation between each prospective user and the broker. Hence, all the SLAs-P that co-exist at a given time interval may have, in general, different values for these parameters. However, it is possible that the broker proposes to its users a predefined set of differentiated service levels, to drive the user indication of a service level, but this does not change the formulation of our problem.

In any case, all the co-existing SLAs-P define the dependability objectives that the broker must meet in that interval, provided that the flow of requests generated by the users in that interval does not exceeds the limits stated by the L values in the SLAs-P. Moreover, they also define the expected income for the broker.

To meet these objectives, we assume that the broker has already identified for each abstract service S_i a pool of corresponding concrete service, negotiating with each of them a SLA-R concerning its dependability and cost, and the load it is able to sustain.

Thus, the set of all these SLA-R defines the constraints within which the broker can organize an adaptation policy.

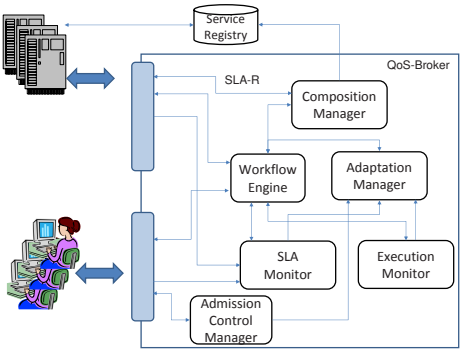


Fig. 2. Broker architecture

2.3 Broker Architecture

To carry out its task, the broker architecture is structured around the following components, as depicted in Fig. 2: the *Composition Manager*, the *Workflow Engine*, the *Adaptation Manager*, the *Execution Monitor*, the *Admission Control Manager* and the *SLA Monitor*. Our general broker architecture is inspired to existing implementation of frameworks for Web services QoS brokering, e.g., [19,20,17]. In what follows, we summarize the respective tasks of the broker architecture components.

Composition Manager. The main functions of the Composition Manager are the service composition (*i.e.*, the specification of the business process, whose structure can be abstractly described by a labeled syntax tree generated by the grammar in Sect. 2.1), the discovery of the component concrete services and the negotiation and establishment of the SLAs with the providers of the concrete services (SLA-R).

Workflow Engine. The Workflow Engine is the software platform executing the business process (e.g., ActiveBPEL or ApacheODE). Once a user has been admitted with an established SLA, the Workflow Engine acts as the broker front-end to the user for the service provisioning. When the user invokes the process, the Workflow Engine creates a new instance of the process itself. Each generated instance can be different, according to the instructions received by the Adaptation Manager (described below). For example, the service request of users having different SLA-P could be bound to different concrete services. Moreover, the workflow structure can also be modified.

Execution Monitor. The Execution monitor collects information about the composite service usage, calculating estimates of the model parameters. In our methodology, the only needed parameters are the invocation frequencies of the functionalities (abstract services) used to build the composite service.

SLA Monitor. The SLA Monitor collects information about the dependability level perceived by the users and offered by the providers of the used component services, and about the mean volume of requests generated by the

users. Furthermore, the SLA Monitor signals whether there is some variation in the pool of service instances available for a given abstract service (i.e., it notifies if some service goes down/is unavailable). In literature, there are examples of SLA Monitors managed/owned by the broker [19,20,17] as well as of third party SLA monitors. In our specific case, we consider an SLA monitor directly managed by the broker.

Admission Control Manager. The Admission Control Manager determines whether a new user can be accepted, given the associated SLA-R, without violating already existing SLA-Ps and SLA-Rs.

The latter three modules (*i.e.*, Execution Monitor, SLA Monitor, and Admission Control Manager) collectively play a two-fold role. On the one hand, they maintain up to date the parameters of the model of the composite service operations and environment. These parameters include the invocation frequencies of the abstract services, the rate of arrival of service requests, the dependability and cost of the used concrete services.

On the other hand, when these modules observe significant variations in the model parameters, they signal these events to the Adaptation Manager. Summarizing, the Admission Control Manager signals events related to the fluctuation of workload intensity parameters (the arrival or departure of users, with the consequent variation in the incoming flow of requests, according to what stated in their SLAs), while the Execution Monitor signals abnormal fluctuation in the composite service usage, and the SLA Monitor signals abnormal events, such as unreachability of a concrete service and variation of its dependability level.

Adaptation Manager. Upon receiving a notification of a significant variation of the model parameters, the Adaptation Manager finds out whether an adaptation action must be performed. To this end, it executes the adaptation algorithm, passing to it the new instance of the system model with the new values of the parameters. The calculated solution provides indications about the adaptation actions that must be performed to optimize the use of the available resources (i.e., the concrete services) with respect to the utility criterion of the broker. Based on this solution, the Adaptation Manager issues suitable directives to the Workflow Engine, so that future instances of the business process will be generated according to these directives. The possible adaptation actions, already outlined in the introduction, are detailed in the following section.

2.4 Adaptation Actions

We recall from the introduction that in our approach the broker deals simultaneously with users having different requirements stated in the corresponding SLAs-P. Each request for the composite service coming from a user generates a corresponding set of one (or more) requests for each abstract service S_i. These latter requests must be bound to suitable concrete services $S_{i.j}, 1 \leq j \leq n_i$.

A first action used by the broker to fulfill the SLAs negotiated with its users is based on *service selection*, which leads to binding each request for S_i to a single

$S_{i.j}$. In most of the current literature on service selection, where single service requests are considered in isolation (e.g.,[21,11]), this action consists in a *0-1 choice* of one concrete service $S_{i.j}$ from the available ones. In our approach, instead, we consider simultaneously all the requests belonging to the flow generated by each service user. Hence, the service selection action consists in determining, for each abstract service S_i, which is the *fraction* of the overall set of requests generated for S_i by a user that will be bound to a given concrete service $S_{i.j}$. Abstractly, this adaptation action can be represented by the introduction of a probabilistic switch in the abstract service S_i, which routes toward the $S_{i.j}$'s requests arriving to S_i according to a suitable set of probabolities, as depicted in Fig. 3(a).

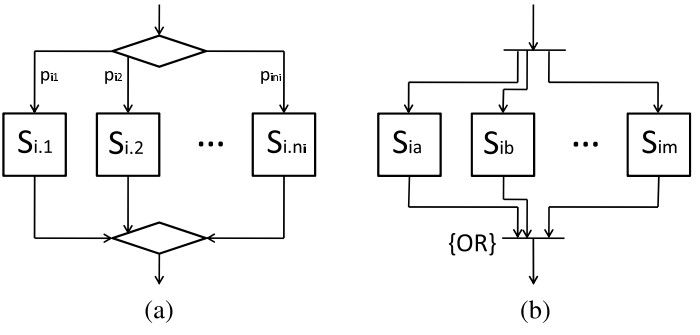


Fig. 3. Adaptation actions

We point out that, as the broker deals simultaneously with several users having different requirements, each user will have its own switch for each S_i. Hence, requests coming from different users will be likely routed differently. For requests coming from the same user, it is possible as a special case that the switch routes all the requests for S_i to a single $S_{i.j}$, but in general the service selection allows to route subsets of these requests to different $S_{i.j}$'s.

As already outlined in the introduction, dynamic service selection is the primary adaptation actions considered in several papers on SOA systems. However, it is possible that a user arrives with high dependability requirements, which cannot be satisfied by any selection of the single concrete services already identified by the broker. Rather than rejecting this user (which could cause an income loss and/or a decrease in the broker reputation), the broker could try other possible actions:

1. To identify additional concrete services, with higher dependability;
2. To "increase" the dependability which can be attained using the already identified concrete services.

The former action has two drawbacks. It requires additional effort to discover such services and negotiate with them suitable SLAs. Worse yet, such services could not even exist.

The latter action does not suffer from these drawbacks. It is based on the idea that the availability of multiple independent implementations $S_{i.1}, S_{i.2}, \ldots, S_{i.n_i}$ of an abstract service S_i naturally suggests the use of *spatial redundancy* techniques to get a dependability increase [22]. According to these techniques, a request for S_i is logically bound to a set of two or more $S_{i.j}$'s, rather than to a single $S_{i.j}$. Different spatial redundancy techniques can be devised, which differ in the way the members of the set are used and in the assumed underlying failure model. In this paper, we consider one of such techniques, which works under the *fail-stop* failure model (*i.e.*, either a service responds correctly when it is invoked, or does not respond at all [1]). According to this technique, a request for S_i is sent in parallel to two or more $S_{i.j}$'s, taking as correct reply the first one which arrives. This increases the likelihood that the functionality associated with S_i is correctly carried out with respect to the case where a single concrete service is used, but at a higher cost, equal to the sum of the costs of all the invoked services. We refer to [22,14,23] for a description of other spatial redundancy techniques which works under the same or different failure models (*e.g.*, Byzantine failures).

The use of this adaptation action represents a form of adaptation based on *architecture selection*, as it basically corresponds to a modification of the workflow architecture, as depicted in Fig. 3(b). In this figure, we see that the simple abstract service S_i is substituted by a composite service $par_{or}(S_{i.a}, S_{i.b}, \ldots)$, where each $S_{ia}, S_{ib}, \ldots$ will be bound to a different concrete service $S_{i.j}$.

In our approach, these two actions (service selection and architecture selection) can be used by the broker to dynamically adapt the SOA system it manages to changes in its operating environment. We remark that these actions can coexist, being used by the broker not only for the requests of different users, but also for different requests of the same user.

In the next section we define formally a mathematical model of the SOA system, and the corresponding optimization problem that can be efficiently solved by the Adaptation Manager component. The calculated solution allows driving the selection of the appropriate adaptation actions.

3 Adaptation Model

Given the grammar proposed in Sect. 2.1, that specifies the structure of the composite service, we can use it to define an instance of the composite service and represent the instance through a labeled syntax tree. The tree leaves constitute the set of abstract services S_i that have been identified by the Composition Manager to build the composite service; we denote by $\mathcal{V}$ this set. Given $\mathcal{V}$, we assume that it is known the usage profile of the composite service for each user k, expressed by the quantities V_i^k, $i \in \mathcal{V}$. V_i^k denotes the average number of times the broker invokes S_i to fulfill a request received from the user $k \in K$, where K is the set of users which have a SLA-P with the broker. Therefore, we can label each leaf of the syntax tree, that represents S_i, with a proper vector $V_i = (V_i^1, \ldots, V_i^{|K|})$ (an example of a labeled syntax tree is shown in Fig. 1). For each S_i, the

Composition Manager identifies (e.g., by using information from service registries) a set $I_i = \{S_{i.1}, \ldots, S_{i.n_i}\}$ of concrete services that implement it.

Each concrete service $S_{i.j}$ is characterized by its own dependability levels and cost. Hence, the SLA-R contracted by the broker with each $S_{i.j}$ is specified by an instance of the general SLA template described in Sect. 2.2 and defined by the tuple $\langle a_{ij}, L_{ij}, c_{ij} \rangle$, where L_{ij} is the average load that the broker has agreed to generate towards $S_{i.j}$. On the other hand, we denote by $\langle A^k_{\min}, L^k, C^k \rangle$ the parameters of the SLA-P concerning a user $k \in K$, where L^k is the agreed volume of requests the user will submit to the broker.

In our broker architecture, the Adaptation Manager is responsible for determining the adaptation strategy which, for the given system model and the current parameters values, optimizes a suitable broker utility function while meeting the users dependability constraints.

For each user, the adaptation strategy consists in determining for each abstract service S_i:

1. The adaptation action (service selection, architecture selection or both) to be used;
2. The fraction of each flow of requests for S_i to be bound to the different concrete services for each considered adaptation action.

We model a given strategy by associating with each user k a vector $\boldsymbol{x}^k = (\boldsymbol{x}^k_1, \ldots, \boldsymbol{x}^k_N)$, where $N = |\mathcal{V}|$ and $\boldsymbol{x}^k_i = [x^k_{iJ}]$, with $J \in \mathcal{P}_i = 2^{I_i} \setminus \emptyset$, *i.e.*, J is a non-empty subset of I_i. Hence, index i of x^k_{iJ} ranges over the set of abstract services, while J ranges over all the non-empty subsets of the concrete services implementing S_i.

For each abstract service S_i, the entries x^k_{iJ} of $\boldsymbol{x}^k_i$ denote the fraction of the user k requests which are bound to the set of concrete services J. We can distinguish two cases:

- $J = \{S_{i.j}\}$, *i.e.*, J is a singleton: in this case, the entry x^k_{iJ} denotes the fraction of requests for S_i to be bound to the single concrete service $S_{i.j} \in I_i$, thus using service selection as adaptation action;
- $J = \{S_{i.j_1}, \ldots, S_{i.j_\ell}\}$, $\ell > 1$, $J \in \mathcal{P}_i$: in this case, the entry x^k_{iJ} denote the fraction of requests for S_i to be bound to the set of concrete services $\{S_{i.j_1}, \ldots, S_{i.j_\ell}\}$, thus using architecture selection as adaptation action (indeed, this action corresponds to the replacement of S_i by $par\_or(S_{i.j_1}, \ldots, S_{i.j_\ell})$).

As an example, consider the case of four concrete services $S_{i.1}, \ldots, S_{i.4}$ for a given service S_i and assume that the strategy $\boldsymbol{x}^k_i$ for a given user k specifies the following values: $x_{i.\{S_{i.1}\}} = 0.3$, $x_{i.\{S_{i.3}\}} = 0.3$, $x_{i.\{S_{i.2}, S_{i.4}\}} = 0.4$ and $x_{i.J} = 0$ otherwise. This strategy implies that 30% of user k requests for service S_i are bound to service $S_{i.1}$, 30% are bound to service $S_{i.3}$ while the remaining 40% are bound to the redundant pair $\{S_{i.2}, S_{i.4}\}$ (see Fig. 4).

The Adaptation Manager determines the values of x^k_{iJ} by solving a suitable optimization problem which takes the following general form (the explicit form of the problem we consider will be detailed in Sect. 4):

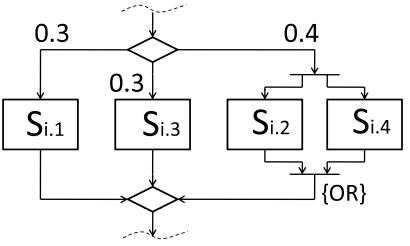


Fig. 4. Example of adaptation strategy

$$\begin{aligned} \textbf{max } & F(\boldsymbol{x}) \\ \textbf{subject to: } & Q^{\alpha}(\boldsymbol{x}) \leq Q^{\alpha}_{\max} \\ & Q^{\beta}(\boldsymbol{x}) \geq Q^{\beta}_{\min} \\ & S(\boldsymbol{x}) \leq L \\ & \boldsymbol{x} \in A \end{aligned} \tag{1}$$

where $\boldsymbol{x} = (\boldsymbol{x}^1, ..., \boldsymbol{x}^{|K|})$ is the decision vector, $F(\boldsymbol{x})$ is a suitable broker objective function, $Q^{\alpha}(\boldsymbol{x})$ and $Q^{\beta}(\boldsymbol{x})$ are, respectively, those QoS attributes whose SLA values are settled as a maximum and a minimum, $S(\boldsymbol{x})$ are the constraints on the offered load determined by the SLAs with the service providers, and $\boldsymbol{x} \in A$ is a set of functional constraints (*e.g.*, this latter set includes the constraint $\sum_{J \in \mathcal{P}_i} x^k_{iJ} = 1$).

A new solution of the optimization problem may be triggered when: *a*) the Execution Monitor identifies some change in the average number of visits to the abstract services; *b*) the service composition changes, because either an abstract service or a concrete service is added or removed; *c*) the SLA Monitor detects some violation in the negotiated SLA parameters; *d*) a new user, which does not have yet a SLA with the broker, asks for the composite service.

The solution of the optimization problem is used by the Adaptation Manager to determine for each invocation of an abstract service S_i the adaptation action to be used - service or architecture selection - and the actual service(es) to implement it by using the vectors $\boldsymbol{x}^k_i$. To bind the requests to the concrete services, the Workflow Engine uses the solution of this optimization problem as follows. Given a user k request, the Workflow Engine considers only the elements of the solution vector $\boldsymbol{x}$ that pertain to a given user k. If, for each abstract service S_i, there is more than one $x^k_{iJ} \neq 0$, the Workflow Engine partitions the flow of requests for S_i among different (subsets of) concrete services, using the x^k_{iJ} values.

4 Optimization Problem

In this section we first present the QoS model for the composite service and how to compute its QoS attributes. We then detail the instance of the general

optimization problem previously outlined. In the following, since dependability measures of a composed service are given the product of the individual dependabilities, we will consider the logarithm of the expected dependability in order to have all aggregate QoS a linear function of the component QoS.

4.1 Composite Service QoS Metrics

For each user $k \in K$, the QoS attributes, namely, the expected logarithm of the dependability $E^k = \log A^k$, which is the logarithm of the probability A^k that the composite service is either available or reliable for a user k request and the expected execution cost C^k, which is the price to be paid by the broker to fulfill a user k request, depend on: 1) the set of concrete services selected to perform each service; and, 2) how the services are orchestrated to provide the composite service.

To compute these quantities, we first compute the expected QoS metric of an abstract service. Let $Z_i^k(\boldsymbol{x})$ denote the QoS attribute of the abstract service $i \in \mathcal{V}$, $Z \in \{E, C\}$ for a strategy $\boldsymbol{x}$. We have $Z_i^k(\boldsymbol{x}) = \sum_{J \in \mathcal{P}_i} x_{iJ}^k z_{iJ}^k$ where z_{iJ}^k, $z \in \{e, c\}$ is function of the QoS attributes of the concrete services in the set J which are used to implement i. We can distinguish two cases, corresponding to the two different adaptation actions that can be used (service selection or architecture selection):

- $\boldsymbol{J = \{S_{i.j}\}}$ *i.e.*, service i is implemented by service $S_{i.j}$, we simply have $e_{iJ} = \log a_{ij}$ and $c_{iJ} = c_{ij}$, *i.e.*, the QoS attribute coincides with that of the selected concrete service $S_{i.j}$;
- $\boldsymbol{J = \{S_{i.j_1}, ..., S_{i.j_\ell}\}}$, $\ell > 1$, service i is implemente by the set $\{S_{i.j_1}, \ldots, S_{i.j_\ell}\}$ in spatial redudancy and we have:

$$e_{iJ} = \log \left(1 - \prod_{h=1}^{\ell} (1 - a_{ij_h}) \right) \tag{2}$$

$$c_{iJ} = \sum_{h=1}^{\ell} c_{ij_h} \tag{3}$$

The expression for e_{iJ}, which represents (the logarithm of) the probability that the composite service *par_or*$(S_{i.j_1}, \ldots, S_{i.j_\ell})$ terminates successfully, is given by (the logarithm of) the complement to one of the probability that all concrete services in J fail. The cost c_{iJ} is simply the sum of the cost of the invoked services, since for each invocation of i all the concrete service in J are invoked.

The QoS attributes for the composed services can be computed by properly aggregating the corresponding QoS attributes of the constituend services i [24]. Since the cost and (logarithm of the) dependability are additive [24] QoS metrics, for their expected value we simply obtain

$$E^k(\boldsymbol{x}) = \sum_{i \in \mathcal{V}} V_i^k E_i^k(\boldsymbol{x}) = \sum_{i \in \mathcal{V}} V_i^k \sum_{J \in \mathcal{P}_i} x_{i.J}^k e_{iJ} \tag{4}$$

$$C^k(\boldsymbol{x}) = \sum_{i \in \mathcal{V}} V_i^k C_i^k(\boldsymbol{x}) = \sum_{i \in \mathcal{V}} V_i^k \sum_{J \in \mathcal{P}_i} x_{iJ}^k c_{i.J} \tag{5}$$

where V_i^k is the expected number of times service i is invoked for a class k request.

4.2 Optimization Model

In this section we detail the instance of the general optimization problem outlined in Sect. 3. By solving this problem, the Adaptation Manager determines the variables x_{iJ}^k, $i \in \mathcal{V}$, $k \in K$, $J \in \mathcal{P}_i$ which maximize a suitable objective function given the user QoS and system constraints.

We assume that the broker wants, in general, to define an adaptation strategy which optimize multiple - possibly conflicting - requirements; therefore, the adaptation strategy results in a multi-objective optimization. We tackle the multi-objective problem by transforming it into a single objective problem through the weighted sum approach, which is the most widely used scalarization method. To this end, we define as the broker utility function $F(\boldsymbol{x})$ the weighted sum of the (normalized) QoS attributes of all users which can be regarded as an overall aggregate QoS measure for the offered service. More precisely, let $Z(\boldsymbol{x}) = \frac{1}{\sum_{k \in K} L^k} \sum_{k \in K} L^k Z^k(\boldsymbol{x})$, where $Z \in \{E, C\}$ is the expected overall dependability and cost, respectively. We define the broker utility function as follows:

$$F(\boldsymbol{x}) = w_e \frac{E(\boldsymbol{x}) - E_{\min}}{E_{\max} - E_{\min}} + w_c \frac{C_{\max} - C(\boldsymbol{x})}{C_{\max} - C_{\min}} \tag{6}$$

where w_e and w_c ($w_e, w_c \geq 0$, $w_e + w_c = 1$) are weights for the different QoS attributes. Here, $E_{\max}$ ($E_{\min}$) and $C_{\max}$ ($C_{\min}$) denote, respectively, the maximum (minimum) value of the aggregated dependability (cost) (We will describe how to determine these values shortly). $F(\boldsymbol{x})$ takes values in the interval $[0, 1]$. Assuming $w_e, w_c \neq 0$, $F(\boldsymbol{x}) = 1$ when $E(\boldsymbol{x}) = E_{\max}$ and $C(\boldsymbol{x}) = C_{\min}$, *i.e.*, when the aggregate dependability is maximized and the cost minimized; $F(\boldsymbol{x}) = 0$ when $E(\boldsymbol{x}) = E_{\min}$ and $C(\boldsymbol{x}) = C_{\max}$, *i.e.*, when the aggregate dependability is minimized and the cost maximized.

The problem solved by the Adaptation Manager consists in finding the variables x_{iJ}^k, $i \in \mathcal{V}$, $k \in K$, $J \in \mathcal{P}_i$, which maximizes the broker utility $F(\boldsymbol{x})$. This is accomplished by solving the following linear optimization problem:

$$\textbf{max } F(\boldsymbol{x})$$

$$\textbf{subject to: } C^k(\boldsymbol{x}) \leq C_{\max}^k \quad k \in K \tag{7}$$

$$E^k(\boldsymbol{x}) \geq E_{\min}^k \quad k \in K \tag{8}$$

$$\sum_{k \in K} \sum_{J \in \mathcal{P}_i} x_{iJ}^k V_i^k L^k \leq L_{ij} \quad i \in \mathcal{V}, j \in I_i \tag{9}$$

$$x_{iJ}^k \geq 0, J \in \mathcal{P}_i(j), \sum_{J \in \mathcal{P}_i} x_{iJ}^k = 1 \quad i \in \mathcal{V}, k \in K \tag{10}$$

Equations (7)-(8) are the QoS constraints. $E_{\min}^k = \log A_{\min}^k$ is the logarithm of user k minimum expected service dependability. $C_{\max}^k$ is the maximum cost the

broker is willing to pay to fulfil a user k request. We assume $C^k_{\max} \leq C^k$, where C^k is the cost the broker charge user k for each service request as defined by user k SLA-P. Equations (9) are the SLA-R constraints and ensure that the broker does not exceed the volume of invocations agreed with the service providers. The left hand side in (9) is indeed the volume of requests which are bound to service $S_{i.j}$ and $L_{i.j}$ the agreed upon value in the SLA-R. Finally, (10) are the functional constraints.

We observe that the proposed optimization problem is a Linear Programming problem which can be efficiently solved via standard techniques. The solution thus lends itself to both on-line and off-line operations. The problem can be extended to account for other QoS attributes, *e.g.*, the service time and reputation (see [13] for details). Is it worth noting that also in these cases the optimization problem can be cast as a Linear Programming problem.

We conclude describing on how to compute the maximum and minimum values of the QoS attributes in the objective function. $E_{\min}$ and $C_{\max}$ are simply expressed respectively in terms of $E^k_{\min}$ and $C^k_{\max}$. For example, the maximum cost is given by $C_{\max} = \frac{1}{\sum_{k \in K} L^k} \sum_{k \in K} L^k C^k_{\max}$. Similar expression holds for $E_{\min}$. The values for $E_{\max}$ and $C_{\min}$ are determined by solving a modified optimization problem in which the objective function is the QoS attribute of interest, subject to the constraints (9)-(10).

5 Numerical Experiments

In this section, we illustrate the behavior of the proposed adaptation strategy scheme through the simple abstract workflow of Fig. 1.

For the sake of simplicity we assume that the broker has established just two SLA-Rs for each service except for service S_2 for which there are 4 established SLA-Rs. The different SLA-Rs differ in terms of cost and dependability. Tables 2 summarizes the parameters as defined by the SLA-R $\langle a_{ij}, c_{ij}, L_{ij} \rangle$ for each concrete service $S_{i.j}$. They have been chosen so that for each abstract service $S_i \in \mathcal{V}$, concrete service $S_{i.1}$ represents the *better* service, which at a higher cost guarantees higher dependability with respect to service $S_{i.2}$, which costs less but has lower dependability. For all services, we assume $L_{ij} = 10$.

In Table 3 we also list the QoS parameters associated with sets of concrete services $S_{i.j}$s used in spatial redundancy according to the fail stop failure model, *i.e.*, the QoS associated to the constrct $par_{or}(S_{i.1}, \ldots, S_{i.\ell})$. The values are computed from those of the constituent services using (2)-(3). We assume the broker has established SLA-Ps with 4 users which are characterized by a wide range of dependability requirements as listed in Table 4. Users are ordered according to the required minimum level of dependability, with User 1 having the highest requirement, $A^1_{\min} = 0.995$, and User 4 the least requirement $A^4_{\min} = 0.9$. The SLA-Ps costs have been set accordingly with User 1 incurring the highest cost per request, $C^1 = 25$, and User 4 only $C^4 = 12$. We consider the following values for the the expected number of service invocations of the different users: $V^k_1 = V^k_2 = V^k_3 = 1.5$, $V^k_4 = 1$, $k \in K$, $V^k_5 = 0.7$, $V^k_6 = 0.3$, $k \in \{1, 3, 4\}$, $k \neq 2$,

Table 2. Concrete services QoS attributes

Serv.	c_{ij}	a_{ij}
$S_{1.1}$	6	0.999
$S_{1.2}$	3	0.99
$S_{2.1}$	4	0.999
$S_{2.2}$	2	0.99
$S_{2.3}$	4.5	0.99
$S_{2.4}$	1	0.95
$S_{3.1}$	2	0.999
$S_{3.2}$	1	0.99
$S_{4.1}$	0.5	0.999
$S_{4.2}$	0.3	0.99
$S_{5.1}$	1	0.999
$S_{5.2}$	0.7	0.99
$S_{6.1}$	0.5	0.999
$S_{6.2}$	0.2	0.99

and $V_5^2 = V_6^2 = 0.5$. In other words, all users have the same average number of service invocations except for user 2, which invokes the services 5 and 6 with different probabilities from the other users. We illustrate the adaptation strategies under two different scenarios: 1) the broker minimizes the average cost ($w_c = 1$); and 2) the broker maximizes the average dependability ($w_e = 1$). The results are summarised in Fig. 5 and 6 which shows the solutions for the two scenarios for User 1 and User 4, respectively; in Table 5 we list the resulting QoS metrics for all the users.

In the first scenario, the broker goal is to minimize the expected cost (which in turn maximizes the broker profit). In this setting the broker has no incentive to guarantee to the user more than the minimum required. As a result the solution provided by the Adaptation Manager guarantees only the minimum required level

Table 3. QoS attributes for the fail stop redundant services

Redundant Serv.	c_{iJ}	e_{iJ}
$par_{or}(S_{1.1}, S_{1.2})$	9	log(0.99999)
$par_{or}(S_{1.1}, S_{1.3})$	6	log(0.99999)
$par_{or}(S_{2.1}, S_{2.3})$	8.5	log(0.99999)
$par_{or}(S_{2.1}, S_{2.4})$	5	log(0.99995)
$par_{or}(S_{2.2}, S_{2.3})$	6.5	log(0.9999)
$par_{or}(S_{2.2}, S_{2.4})$	3	log(0.9995)
$par_{or}(S_{2.3}, S_{2.4})$	5.5	log(0.9995)
$par_{or}(S_{2.1}, S_{2.2}, S_{2.3})$	10.5	log(0.9999999)
$par_{or}(S_{2.1}, S_{2.2}, S_{2.4})$	7	log(0.9999995)
$par_{or}(S_{2.1}, S_{2.3}, S_{2.4})$	9.5	log(0.9999995)
$par_{or}(S_{2.2}, S_{2.3}, S_{2.4})$	7.5	log(0.999995)
$par_{or}(S_{2.1}, S_{2.2}, S_{2.3}, S_{2.4})$	11.5	log(0.999999995)
$par_{or}(S_{3.1}, S_{3.2})$	3	log(0.99999)
$par_{or}(S_{4.1}, S_{4.2})$	0.8	log(0.99999)
$par_{or}(S_{5.1}, S_{5.2})$	1.7	log(0.99999)
$par_{or}(S_{6.1}, S_{6.2})$	0.7	log(0.99999)

Table 4. User SLA-P attributes

User	$A^k_{\min}$	C^k	$L^k_{\max}$
1	0.995	25	1.5
2	0.99	20	1
3	0.95	15	3
4	0.9	12	1

Table 5. User QoS metrics

User	E^k	C^k
1	log(0.995)	19.08
2	log(0.99)	17.28
3	log(0.95)	10.21
4	log(0.9)	8.85

Scenario 1 ($w_c = 1$)

User	E^k	C^k
1	log(0.9991)	25
2	log(0.9976)	20
3	log(0.9828)	15
4	log(0.9684)	12

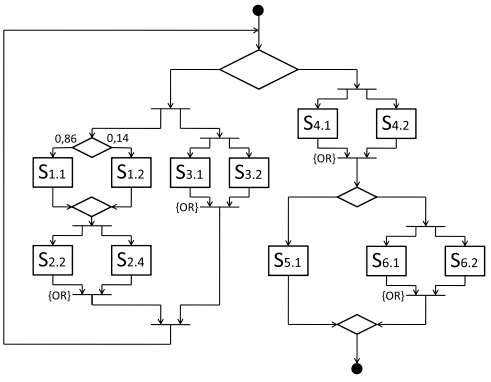
Scenario 2 ($w_e = 1$)

of dependability, *i.e.*, $E^k(\boldsymbol{x}) = E^k_{\min} = \log A^k_{\min}$ (see Table 5 (left)) with increasing costs with the level of dependability. For user 4, this results in a workflow with no form of redundancy (see Fig. 5) and where most of the services are deterministically bound to the cheaper $S_{i.2}$ services. Observe that the solution still requires the use of the more expensive concrete service $S_{4.1}$ and a combination of $S_{2.2}$ and $S_{2.4}$ since otherwise the minimimum level of dependability required by the SLA-P could not be met. The solution for user 1 differs substantially from the one just described. User 1 workflow is characterized by spatial redundancy for most of the services. This comes at a significant higher cost per request (19.08 more than twice the 8.85 needed to satisfy a user 4 request).

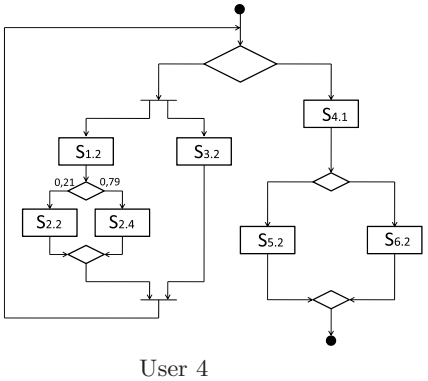
We now turn our attention to the second scenario, *i.e.*, where the broker goal is to maximize the users' dependability. In this setting, the solution provided by the Adaptation Manager is bounded by the resources available to implement the services, *i.e.*, the service providers, and by the maximum cost the broker is will to pay for each user (which defines its profit margin). Here for the sake of simplicity, we assume $C^k_{\max} = C^k$. From Table 5 (right) we see that the optimal solution is achieved by maximizing the cost the broker pays per request since $C^k(\boldsymbol{x}) = C^k_{\max}$ for all users. This in turn guarantees significantly higher level of dependability than those requested by the users. From Fig. 6 we see that for both users the dependability increase is achieved by using better services, *e.g.*, in user 4 solution $S_{5.2}$ and $S_{6.2}$ are replaced by $S_{5.1}$ and $S_{6.1}$, respectively, and redundancy, *e.g.*, service S_2 for user 1 and service S_5 and S_6 for user 4.

6 Related Work

As outlined in [5], the topic of self-adaptive systems has been studied in several communities such as distributed systems, biologically-inspired computing,



User 1



User 4

Fig. 5. Adaptation Manager solution: scenario 1 - cost minimization

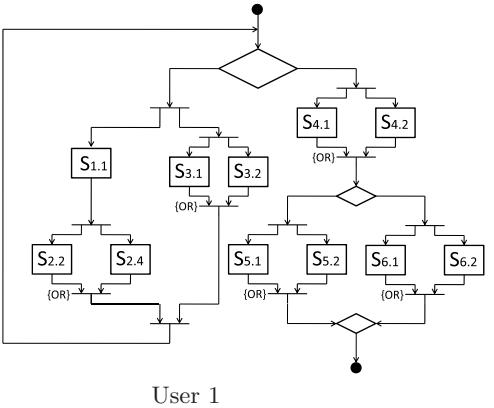
robotics, machine learning, control theory, network-based systems, etc. and recently also in the software engineering field [5,25]. In particular, approaches spanning software architecture [8], service-oriented applications [11,9], pervasive applications [26] and autonomic systems [4] have been recently proposed.

In the area of *autonomic computing*, the original approach proposed by IBM [7] was an architecture-level approach in which the generic architecture of an autonomic system was defined as a system composed by managers and managed resources. In this approach the manager communicates with the resource through a sensor/actuator mechanism and the decision is elaborated using the so-called MAPE-K (Monitor, Analyze, Plan, Execute and Knowledge) cycle. This loop collects information from the system, makes decisions and then organizes the actions needed to achieve goals and objectives, and controls the execution. All the manager's functions consume and generate knowledge, which is continuously shared leading to better-informed decisions. From this perspective, the architecture we have outlined in section 2, and the adaptation methodology discussed

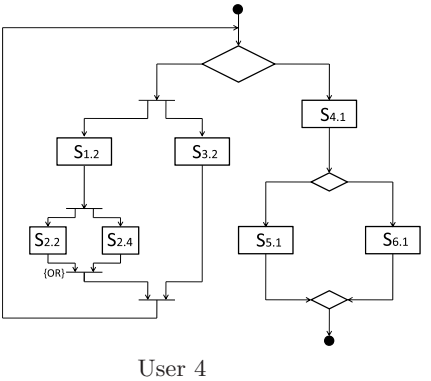
in sections 3 and 4 can be seen as an instantiation for the SOA environment of an autonomic system, focused on the fulfillment of dependability requirements. In particular, the Execution Monitor, SLA Monitor and Admission Control Manager collectively implement the Monitor and Analyze functions, while the Adaptation Manager and Workflow Engine implement the Plan and Execute functions.

Hereafter, we focus on works appeared in the literature dealing with issues concerning the dependability evaluation and the self-adaptation of SOA systems, to guarantee the fulfillment of dependability requirements.

The dependability of a SOA system is difficult to achieve because in the SOA environment the system components are autonomous, heterogeneous and usually come from different providers. In traditional software engineering, many software reliability and availability models have been presented to solve this problem (e.g., [27]). Unfortunately, these models cannot be directly applied to service-oriented systems, where users and providers are distributed, and the processes of service publication, search and invoking are separated [9,10,28]. Indeed, in these systems, an execution failure can be observed for reasons related both to the



User 1



User 4

Fig. 6. Adaptation Manager solution: scenario 2 - dependability maximization

execution environment (e.g., variations in the execution environment configuration, overloaded resource conditions, system running out of memory) or to the non-availability of required services or software components [28,29].

A basic problem to be solved when dealing with QoS issues of SOA systems, is how to determine the QoS attributes of a composite system, given the QoS delivered by its component services. Papers that provide some methods to derive QoS related measures of workflow processes are, for example, [30,31,32].

Another problem, which is also the main focus of this paper, concerns the dynamic adaptation of a SOA system to meet the QoS requirements in a volatile operating environment. Two main classes of approaches have been proposed in the literature to deal with this problem, as already discussed in the introduction. The first one includes approaches mainly based on QoS-based service selection methods as adaptation mechanism. In this case new service components are selected to deal with changes in the operating scenarios. The second class includes approaches based on architecture selection mechanisms, where the adaptation to changes is performed defining new (redundancy based) architectures for the service composition to meet the QoS (basically, dependability) requirements.

Early proposals for dynamic adaptation based on service selection considered only *local* constraints (i.e., constraints which can pose restrictions only on the execution of individual abstract services). In that case, the service selection is very simple and can be performed at run time by a greedy approach that selects the best candidate service suitable for the execution [33]. More recent solutions support also *global* constraints [34,21,35,11], adopting a *per-request* approach. Zeng et al. [21], for example, present a global planning approach to select an optimal execution plan by means of integer programming. They propose a simple QoS model using the attributes: price, availability, reliability, and reputation; and then they apply linear programming for solving the optimization QoS matrix formed by all of the possible execution plans to obtain the maximum QoS values. Ardagna and Pernici [11] model the service composition as a mixed integer linear problem where both local and global constraints are taken into account. Their approach is formulated as an optimization problem handling the whole application instead of each execution path separately. Canfora et al. [35] adopt a quite different strategy for optimal selection based on genetic algorithms. An iterative procedure is defined to search for the best solution of a given problem among a constant size population without the need for linearization required by integer programming.

The approaches presented in [36,13,37] differ from previous works that have tackled the service selection as an optimization problem in that the optimization is performed on a per-flow rather than per-request basis. In these approaches the solution of the optimization problem holds for all the requests in a flow, and is recalculated only when some significant event occurs (e.g., a change in the availability or the QoS values of the selected concrete services). Moreover, the optimization problem is solved taking into account simultaneously the flows of requests generated by multiple users, with possibly different QoS constraints.

Considering the architecture selection approaches, to the best of our knowledge few methodologies have been proposed to dynamically determine the most suitable (redundancy based) architecture in a given operating environment. The paper by Guo and others [10] provides a methodology to select different redundancy mechanisms to improve the dependability experienced by a single request addressed to a composite service. The selection problem is formulated as a mixed integer programming problem, and some heuristics are proposed to calculate in an efficient way an approximate solution. An analogous problem is considered in [38]. The proposed methodology is motivated by its application to component-based systems, but it can be easily extended to a SOA environment.

With respect to these methodologies, the framework we propose intends to consider the use of both service selection and architecture selection as adaptation mechanisms, to increase the flexibility of the Adaptation Manager. Differently from most of the papers cited above, we consider an adaptation scenario concerning multiple concurrent flows of requests generated by different users, rather than a single requests. We have discussed in the introduction pros and cons of the per-flow rather than per-request approaches. In this respect, the methodology presented in this paper is an extension of methodologies presented in [36,13,37] as it also considers architecture selection besides service selection as adaptation mechanism.

Differently from methodological papers on architecture selection as adaptation mechanism, more papers exist dealing with issues concerning the implementation of this mechanism. Several approaches have been proposed in the area of Grid applications (see [14] for a survey on approaches for building and executing workflows on Grids) and also applied in the area of service-based systems. These methods are mainly based on retry and redundancy techniques. The retry technique simply tries to execute the same task on the same resource after failure, while in the redundancy approaches it is assumed that there is more than one implementation for a certain computation with different execution characteristics. The problem of Web Service replication has been tackled by Salas et al. in [39] by proposing an infrastructure for WAN replication of Web Services. A different approach, based on a middleware that supports reliable Web Services built on active replication has been proposed in [40]. Similarly, Erradi et al. [41] propose a lightweight service-oriented middleware for transparently enacting recovery action in service-based processes; and Charfi et al. [42] use an aspect-based container to provide middleware support for BPEL that plugs in support for non-functional requirements. Chen et al. [43] construct composite services resilient to various failure types using inherent redundancy and diversity of Web Service components jointly with mediator approach. A different set of works proposes language-based approaches dealing with workflow adaptability through the introduction of additional language constructs. BPEL for Java (BPELJ), for example, combines the capabilities of BPEL and the Java programming language [44]; in [45], Ezenwoye et al. propose a language-based approach to transparently adapt BPEL processes to address reliability at the business process layer. Baresi et al. in [46] propose an approach where BPEL

processes are monitored at run-time through aspect-oriented techniques to check whether individual services comply with their contracts.

With respect to our framework, these proposals can provide useful suggestions about the implementation of the considered adaptation mechanisms.

7 Conclusions

We have presented an approach towards the realization of a SOA system able to self-adapt in a dynamically changing environment, to meet the dependability requirements of several classes of users. We have discussed a possible architecture for this system, which can be seen as an instantiation for the SOA environment of the general architectural framework for self-adapting systems proposed within the autonomic computing initiative. Given this architecture, we have focused on the problem of determining suitable adaptation actions in response to detected environment changes. In this respect, the basic guideline we have followed has been to give a high degree of flexibility to the Adaptation Manager, to meet a broader range of dependability requirements in different operating environments. For this purpose, our methodology allows to adopt simultaneously (for different users, but also for different requests generated by the same user) adaptation actions based on the two main approaches proposed in the literature, called in this paper service selection and architecture selection, respectively.

The proposed approach represents a first step that needs refinements and extensions in several directions. With regard to the adaptation mechanisms, we have actually considered just one kind of adaptation based on the architecture selection paradigm: the replacement of a single service with the "parallel-or" of different implementations of that service. Other kinds of adaptation mechanisms could be considered, using in different ways the spatial redundancy concept: e.g., sequential retry or majority voting. Considering also these mechanisms would increase the flexibility of the Adaptation Manager, as they allow to achieve different cost/benefit tradeoffs, and/or to deal with different failure scenarios. Moreover, a greater flexibility would also be achieved by broadening the scope of the "dependability" concept: in this paper we have limited our attention to the reliability and availability attributes, but we could consider a more general definition of dependability as ability of fulfilling a given set of QoS requirements, which could include other attributes like performance or reputation. We are currently working towards an extension of our methodology along this direction.

With regard to the methodology we have proposed, we point out that a potential problem could be caused by the high number of variables x_{iJ}^{k} in the optimization problem, for high numbers of abstract services and concrete services implementing them. In this case, a possible way to alleviate the problem could be to limit the number of considered subsets J to those having at most a given cardinality (e.g., three), considering the diminishing dependability increase we can achieve with higher redundancy levels.

Finally, we have not dealt in depth with issues concerning the implementation of the adaptation methodology. As pointed out in the related work section,

several proposals exist in the literature, which provide useful contributions in this direction. Based on them, we are working towards the implementation of a prototype to validate our methodology through real experiments.

Acknowledgments

Work partially supported by the Italian PRIN project D-ASAP and by the project Q-ImPrESS (215013) funded under the European Union's Seventh Framework Programme (FP7).

References

1. Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Trans. Dependable Sec. Comput. 1(1), 11–33 (2004)
2. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: Issue and challenges. IEEE Computer 39(10), 36–43 (2006)
3. Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M.P., Pohl, K.: A journey to highly dynamic, self-adaptive service-based applications. Autom. Softw. Eng. 15 (3-4), 313–341 (2008)
4. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing - degrees, models, and applications. ACM Comput. Surv. 40(3) (2008)
5. Cheng, B.H.C., Giese, H., Inverardi, P., Magee, J., de Lemos, R.: 08031 – software engineering for self-adaptive systems: A research road map. In: Software Engineering for Self-Adaptive Systems. Dagstuhl Seminar Proceedings, IBFI, Schloss Dagstuhl, vol. 08031 (2008)
6. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. IEEE Computer 37(7), 56–64 (2004)
7. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
8. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Future of Software Engineering 2007, pp. 259–268 (2007)
9. Birman, K.P., van Renesse, R., Vogels, W.: Adding high availability and autonomic behavior to web services. In: ICSE 2004, pp. 17–26 (2004)
10. Guo, H., Huai, J., Li, H., Deng, T., Li, Y., Du, Z.: Angel: Optimal configuration for high available service composition. In: ICWS 2007, pp. 280–287 (2007)
11. Ardagna, D., Pernici, B.: Adaptive service composition in flexible processes. IEEE Trans. Softw. Eng. 33, 369–384 (2007)
12. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. SIGOPS Oper. Syst. Rev. 41(6), 205–220 (2007)
13. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F.: Flow-based service selection for web service composition supporting multiple qos classes. In: ICWS 2007, pp. 743–750. IEEE Computer Society, Los Alamitos (2007)
14. Yu, J., Buyya, R.: Taxonomy of workflow management systems for grid computing. Journal of Grid Computing 3(3-4) (2005)
15. OASIS: Web Services Business Process Execution Language Version 2.0 (2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`

16. The OWL Services Coalition: OWL-S: Semantic Markup for Web Services (2003), http://www.daml.org/services/owl-s/1.0/owl-s.pdf
17. Dan, A., Davis, D., Kearney, R., Keller, A., King, R., Kuebler, D., Ludwig, H., Polan, M., Spreitzer, M., Youssef, A.: Web services on demand: WSLA-driven automated management. IBM Systems J. 43(1) (2004)
18. Toktar, E., Pujolle, G., Jamhour, E., Penna, M.C., Fonseca, M.: An XML model for SLA definition with key indicators. In: Medhi, D., Nogueira, J.M.S., Pfeifer, T., Wu, S.F. (eds.) IPOM 2007. LNCS, vol. 4786, pp. 196–199. Springer, Heidelberg (2007)
19. Menascé, D.A., Ruan, H., Gomaa, H.: QoS management in service oriented architectures. Performance Evaluation J. 7-8(64) (2007)
20. Nan, Z., Qiu, X.-S., Meng, L.-M.: A SLA-based service process management approach for SOA. In: ChinaCom 2006, pp.1–6 (2006)
21. Zeng, L., Benatallah, B., Dumas, M., Kalagnamam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Trans. Soft. Eng. 30(5) (2004)
22. Chan, P.P.W., Liu, M.R., Malek, M.: Reliable web services: methodology, experiment and modeling. In: ICWS 2007, pp. 679–686. IEEE Computer Society, Los Alamitos (2007)
23. Kotla, R., Clement, A., Wong, E., Alvisi, L., Dahlin, M.: Zyzzyva: speculative byzantine fault tolerance. Communications of the ACM 51(11), 86–95 (2008)
24. Cardoso, J., Sheth, A.P., Miller, J.A., Arnold, J., Kochut, K.J.: Modeling quality of service for workflows and web service processes. Web Semantics J. 1(3) (2004)
25. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: ICSE 2006, pp. 371–380. ACM, New York (2006)
26. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: framework, approaches, and styles. In: ICSE 2008 Companion, pp. 899–910. ACM, New York (2008)
27. Lyu, M.R.: Software reliability engineering: A roadmap. In: FOSE 2007, pp. 153–170. IEEE Computer Society, Los Alamitos (2007)
28. Goeschka, K.M., Froihofer, L., Dustdar, S.: What SOA can do for software dependability. In: Supplementary Volume of DSN 2008 (2008)
29. Immonen, A., Niemelä, E.: Survey of reliability and availability prediction methods from the viewpoint of software architecture. Software and System Modeling 7(1), 49–65 (2008)
30. Cardoso, J.: Complexity analysis of BPEL web processes. Software Process: Improvement and Practice 12(1), 35–49 (2007)
31. Rud, D., Schmietendorf, A., Dumke, R.: Performance modeling of ws-bpel-based web service compositions. In: Services Computing Workshops, pp. 140–147. IEEE Computer Society, Los Alamitos (2006)
32. Marzolla, M., Mirandola, R.: Performance prediction of web service workflows. In: Overhage, S., Szyperski, C., Reussner, R., Stafford, J.A. (eds.) QoSA 2007. LNCS, vol. 4880, pp. 127–144. Springer, Heidelberg (2008)
33. Maamar, Z., Sheng, Q.Z., Benatallah, B.: Interleaving web services composition and execution using software agents and delegation. In: WSABE 2003 (2003)
34. Yu, T., Zhang, Y., Lin, K.-J.: Efficient algorithms for web services selection with end-to-end qos constraints. ACM Trans. Web 1(1), 1–26 (2007)
35. Canfora, G., di Penta, M., Esposito, R., Villani, M.L.: QoS-aware replanning of composite web services. In: ICWS 2005, pp. 121–129. IEEE Computer Society, Los Alamitos (2005)

36. Cardellini, V., Casalicchio, E., Grassi, V., Mirandola, R.: A framework for optimal service selection in broker-based architectures with multiple QoS classes. In: Services Computing Workshops, pp. 105–112. IEEE Computer Society, Los Alamitos (2006)
37. Ardagna, D., Ghezzi, C., Mirandola, R.: Model driven qos analyses of composed web services. In: Mähönen, P., Pohl, K., Priol, T. (eds.) ServiceWave 2008. LNCS, vol. 5377, pp. 299–311. Springer, Heidelberg (2008)
38. Grosspietsch, K.E.: Optimizing the reliability of component-based n-version approches. In: IPDPS 2002. IEEE Computer Society, Los Alamitos (2002)
39. Salas, J., Perez-Sorrosal, F., Patiño-Martínez, M., Jiménez-Peris, R.: Ws-replication: a framework for highly available web services. In: WWW, pp. 357–366. ACM, New York (2006)
40. Ye, X., Sheni, Y.: A middleware for replicated web services. In: ICWS 2005, IEEE Computer Society, Los Alamitos (2005)
41. Erradi, A., Maheshwari, P.: wsBus: QoS-aware middleware for reliable web services interactions. In: EEE, pp. 634–639. IEEE Computer Society, Los Alamitos (2005)
42. Charfi, A., Mezini, M.: Aspect-oriented workflow languages. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 183–200. Springer, Heidelberg (2006)
43. Chen, Y., Romanovsky, A.: Improving the dependability of web services integration. IT Professional 10(3), 29–35 (2008)
44. IBM, BEA Systems: (BPELJ: BPEL for Java technology) `http://www.ibm.com/developerworks/library/specification/ws-bpelj/`
45. Ezenwoye, O., Sadjadi, S.M.: A language-based approach to addressing reliability in composite web services. In: SEKE, Knowledge Systems Institute Graduate School, pp. 649–654 (2008)
46. Baresi, L., Ghezzi, C., Guinea, S.: Smart monitors for composed services. In: ICSOC 2004, pp. 193–202. ACM, New York (2004)