MOSES: a Framework for QoS Driven Runtime Adaptation of Service-oriented Systems

Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi,
Stefano Iannucci, Francesco Lo Presti, Raffaela Mirandola

Abstract—Architecting software systems according to the service-oriented paradigm, and designing runtime self-adaptable systems are two relevant research areas in today's software engineering. In this paper we address issues that lie at the intersection of these two important fields. First, we present a characterization of the problem space of self-adaptation for service-oriented systems, thus providing a frame of reference where our and other approaches can be classified. Then, we present MOSES, a methodology and a software tool implementing it to support QoS-driven adaptation of a service-oriented system. It works in a specific region of the identified problem space, corresponding to the scenario where a service-oriented system architected as a composite service needs to sustain a traffic of requests generated by several users. MOSES integrates within a unified framework different adaptation mechanisms. In this way it achieves a greater flexibility in facing various operating environments and the possibly conflicting QoS requirements of several concurrent users. Experimental results obtained with a prototype implementation of MOSES show the effectiveness of the proposed approach.

Index Terms—Service-oriented architecture, runtime adaptation, quality of service.

1 Introduction

1.1 Motivation

Two of the major current trends in software engineering are: the increasingly central role of the *service-oriented architecture* (SOA) paradigm in the development of software systems, and the emphasis given to the need of introducing *self-adaptation* features within software systems.

The SOA paradigm encourages the realization of new software systems by composing network-accessible loosely-coupled services. It has its roots in the existence of a widely deployed internetworking infrastructure, and in the general shift that has occurred in the way enterprises operate, where fully integrated enterprises are being replaced by more agile networks of enterprises, offering each other specialized services. According to the SOA paradigm, the development focus shifts from activities concerning the in-house custom design and implementation of the system components, to activities concerning the identification, selection, and composition of services offered by third parties.

The goal of self-adaptation is to alleviate the management problem of complex software systems that operate in highly changing and evolving environments. Such systems should be able to dynamically adapt themselves to their environment with little or no human intervention, in order to meet both functional requirements concerning the overall logic to be implemented and non-functional requirements concerning the quality of service (QoS) levels that should be guaranteed.

The two fields outlined above are quite strictly intertwined. On one hand, SOA-based systems represent a typical domain where self-adaptation can give significant gains. Indeed, the open and dynamic world of services is characterized by a continuous evolution: providers may modify the exported services; new services may become available; existing services may be discontinued by their providers; usage profiles may change over time due to the open market in which they are situated [6]. On the other hand, the loose coupling, dynamic selection and binding features of SOA systems make them particularly amenable to the introduction of runtime adaptation policies. In particular, the use of self-adaptation to fulfill non-functional QoS requirements such as performance, reliability and cost plays a central role in the SOA domain. Indeed, in the envisaged service marketplace (e.g., [44], [48], [51]), several competing services may co-exist implementing the same functionality with different QoS and cost. Thus, a prospective user could choose the services that best suit his/her QoS requirements. Hence, being able to effectively deliver and guarantee the QoS level required by a given class of users may bring competitive advantage to a service provider over the others.

1.2 Contribution

The approach proposed in this paper spans over the two fields summarized above. Our goal is to address

- *Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Stefano Iannucci and Francesco Lo Presti are with University of Roma "Tor Vergata", Italy. E-mails: {cardellini, casalicchio, vgrassi, iannucci, lopresti}@disp.uniroma2.it*
- *Raffaela Mirandola is with Politecnico di Milano, Italy, E-mail: mirandola@elet.polimi.it*
- *The original publication is available at http://dx.doi.org/10.1109/TSE.2011.68 in IEEE Transactions on Software Engineering, Vol. 38, No. 5, pp. 1138-1159, Sept./Oct. 2012.*

issues concerning the design and implementation of a self-adaptive SOA system aimed at maintaining some specified QoS and cost requirements.

General discussions concerning the issues and the state of the art in the design and implementation of self-adaptive software systems have been presented, *e.g.*, in [29], [55], [17], [3], [36], [10], [50]. These papers evidence the number and the several facets of the problems to be tackled. As a consequence, it is unlikely that a single methodology, design or implementation approach could be able to encompass effectively all of them.

In this respect, as a first contribution of this paper, we outline a characterization of the problem space of QoS-driven self-adaptation for the SOA domain, providing a frame of reference for existing literature. Moreover, this characterization can help in the identification of interesting problems arising in different regions of this space and promising ways to tackle them.

Then, as a second contribution (that constitutes the most relevant part of this paper), we present MOSES (*MOdel-based SElf-adaptation of SOA systems*), a methodology and a software tool that implements it, for QoS-driven runtime adaptation of SOA systems. MOSES is tailored for a significant region of the overall problem space, corresponding to the scenario where a SOA system architected as a composite service needs to sustain a traffic of requests generated by several classes of services. Within this scenario, MOSES determines the most suitable configuration of this system for a given operating environment by solving an optimization problem (that is a LP problem), derived from a model of the composite service and of its environment. The adopted model allows MOSES to integrate in a unified framework both the selection of the set of concrete services to be used in the composition, and (possibly) the selection of the coordination pattern for multiple functionally equivalent services, where the latter allows to achieve QoS levels that could not be achieved by using single services. In this respect, MOSES is a step forward with respect to most existing approaches for runtime SOA systems adaptation, that limit the range of their actions to the selection of single services to be used in the composition. We assess the effectiveness of MOSES through an extensive set of experiments performed using the software tool that implements it.

This paper integrates and extends the basic elements of the MOSES methodology and prototype presented in [14] and [7]. Specifically, with respect to those works, the new contributions of this paper can be summarized as follows: (*i*) we have defined a characterization of the problem space of QoS-based self-adaptation for the SOA domain; (*ii*) we have included the management of stateful services; (*iii*) we have implemented an improved version of the MOSES prototype; (*iv*) we have run a thorough set of experiments to validate the whole MOSES methodology and compared the computational cost of MOSES with that of other state of the art approaches.

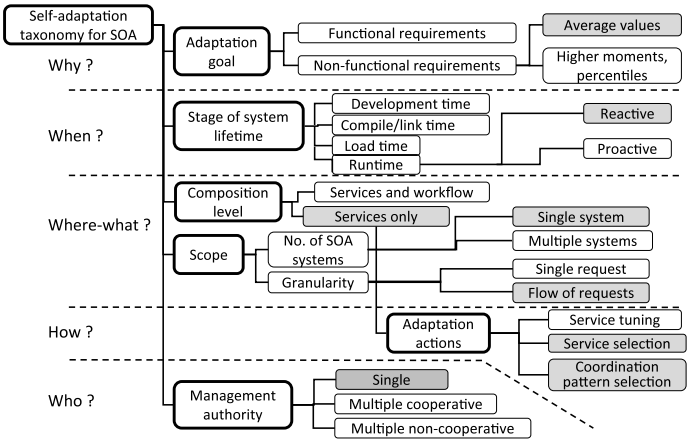


Fig. 1. Taxonomy of self-adaptation for SOA.

1.3 Organization

The remainder of the paper is organized as follows. In Section 2 we examine the problem of self-adaptation from the perspective of the SOA domain, and identify different dimensions that can be used to characterize the problem space. Based on this characterization, we specify in the same section the problem addressed by MOSES. In Section 3 we present an overview of the MOSES framework, and also outline its architecture and the main tasks of its components. Sections 4, 5, and 6 describe specific aspects of MOSES. In particular, in Section 4 we present the adaptation policy model and the QoS model used by MOSES to calculate the overall QoS of a service composition. Based on these models, in Section 5 we present the formulation of an optimization problem that is solved within the MOSES framework to determine a suitable adaptation policy. Then, in Section 6 we describe the MOSES prototype that implements the overall methodology and present a broad set of experiments to assess the effectiveness of the approach and to illustrate the kind of adaptation directives issued by MOSES. In Section 7 we discuss related works. Finally, in Section 8 we summarize some lessons learned with the development of the MOSES methodology, and present directions for future work.

2 Problem Space Characterization

A sensible way to characterize the problem space for self-adaptive software systems is to organize it along several *dimensions*, where each dimension captures one or more related facets of the problem. Papers addressing this issue have provided somewhat different characterizations [29], [55], [17], [3], [36], [10], [50], mainly because of some difference in the adopted perspective. Overall, they can be considered as possible answers to some basic questions [55]:

- *why* should adaptation be performed (which are its goals);
- *when* should adaptation actions be applied;

- *where* the adaptation should occur (in which part of the system) and *what* elements should be changed;
- *how* should adaptation be implemented (by means of which actions);
- *who* should be involved in the adaptation process.

The answers provided by the papers cited above aim at addressing the whole software systems domain. In this section we adopt a narrower viewpoint, and outline possible answers to these questions based on the specific features of the SOA domain, with special emphasis on QoS aspects. We remark that our main goal is to show some of the key issues to be tackled rather than presenting an exhaustive analysis of the literature for the SOA domain.

Figure 1 summarizes the main concepts of this characterization. For the sake of clarity, the class diagram in Fig. 2 illustrates some elements of the SOA domain we use in this characterization. A more detailed taxonomy of these elements can be found, for example, in [9], [19].

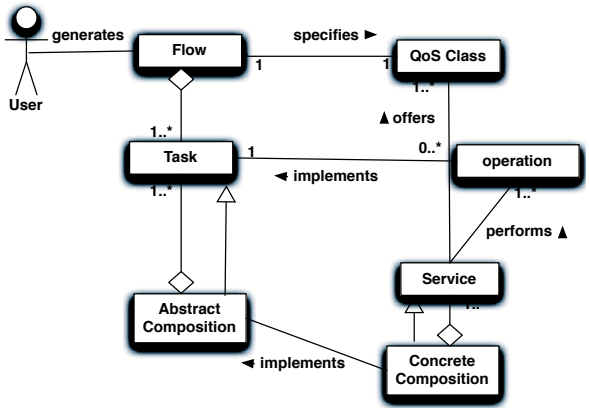


Fig. 2. Conceptual model of the SOA domain.

2.1 Dimensions of Self-adaptation for SOA Systems

Why. The basic goal of adaptation is to make the system able to fulfill its *functional* and/or *non functional* requirements, despite variations in its operating environment, which are very likely to occur in the SOA domain. As pointed out in the introduction, our focus in this paper is on non functional requirements concerning the delivered QoS and cost. In the SOA domain, these requirements are usually the result of a negotiation process engaged between the service provider and user, which culminates in the definition of a *Service Level Agreement* (SLA) concerning their respective obligations and expectations [39]. In a stochastic setting, a SLA specifies guarantees about the *average value* of quality attributes, or more tough guarantees about the *higher moments* or *percentiles* of these attributes.

With regard to functional requirements, we just mention that, in the SOA domain, adaptation may play a relevant role in tackling runtime interoperability issues among dynamically discovered and selected services (e.g., [18], [43]).

When. Broadly speaking, adaptation can be performed at different stages of the system lifetime [36]: development time, compile/link time, load time, runtime. In the SOA domain, the emphasis is on building systems by late composition of running services. Hence, the focus of adaptation in this domain is on the *runtime stage*. This narrower viewpoint of the "when" dimension is also adopted in [55] for the broader field of self-adaptive software. Within this stage, we may further distinguish *reactive* and *proactive* adaptation. In the reactive mode, the system adapts itself after a change has been detected. In the proactive mode, the system anticipates the adaptation based on a prediction of possible future changes.

Where-What. The SOA paradigm emphasizes a compositional approach to software systems development, where the units of composition are services. A service can be considered as a black-box component deployed on some platform, operated by an independent authority and made accessible through some networking infrastructure using standard protocols. Hence, the composition of services can be considered as the basic *locus* for adaptation in the SOA domain. Looking at service composition, we may distinguish an *abstract composition*, where only the required functionalities (*tasks*) and their composition logic are specified, and a *concrete composition*, where the tasks of an abstract composition are bound to actual implementations, based on the use of *operations* offered by network accessible *concrete services*. Based on this distinction, adaptation in the SOA domain may take place at two different levels:

- *services only*: the adaptation only involves the concrete composition, acting on the implementation each task is bound to, leaving unchanged the composition logic (*i.e.*, the overall abstract composition);
- *services and workflow*: the adaptation involves both the concrete and abstract composition; in particular, the composition logic can be altered.

We may also look at the *where-what* question from the perspective of the adaptation *scope*. In this perspective, we may take two different viewpoints: the *number* of SOA systems operating in the same environment that are directly involved in the adaptation process, and the *granularity* level at which adaptation is performed, considering the flow of requests addressed to a SOA system by the same or different users.

We first discuss this issue from the "granularity level" viewpoint in the "scope" dimension:

- *single request*: the adaptation concerns a single service request, and aims at making the system able to fulfill the requirements of that request, irrespective of whether it belongs to some flow generated by one or more users;
- *flow of requests*: the adaptation concerns an overall flow of requests, and aims at fulfilling requirements concerning the global properties of that flow.

Let us consider now the "number of SOA systems" viewpoint:

- *single system*: a single system is explicitly considered as the system to be adapted, while everything else, including other competing SOA systems, is considered part of its environment;
- *multiple systems*: several SOA systems, competing for overlapping sets of services in the same environment, are explicitly considered in the adaptation process.

How. Possible answers to this question depend on the level of the composition where adaptation takes place, as discussed above. For adaptations involving only the services of the composition, adaptation actions could be based on:

- *service tuning*: the behavior and/or properties of the operations of concrete services are changed, depending on the current operating conditions, exploiting some management interface exposed by the concrete services themselves (*e.g.*, based on WSDM MOWS [34]). This kind of action does not change the current binding between tasks and operations of concrete services;
- *service selection*: the goal of this action is to identify and to bind to each task a corresponding *single* operation offered by a concrete service, selecting it from a set of candidates. This kind of action could change the binding between tasks and operations, if the previous selection is no longer suitable for the new operating conditions;
- *coordination pattern selection*: rather than binding each task to a single operation, this action binds it to a *set* of functionally equivalent operations offered by different concrete services, coordinating them according to some spatial or temporal redundancy pattern. The coordination pattern is selected within a set of implementable patterns (e.g., 1-out-of-n parallel redundancy, alternate service), that could in general guarantee different QoS and cost levels, for the same set of coordinated operations. Binding a task to a set of equivalent operations allows to obtain QoS levels (concerning reliability and, in some cases, performance) that could not be achievable binding it to a single operation. Of course this advantage should be weighted against the higher cost caused by the use of multiple concrete services.

Who. This dimension concerns the "authorities" that manage the adaptation process and it is related to the "number of SOA systems" dimension discussed above. In the case of a single system, we may assume that its adaptation is under the control of a *single authority* (that must take into account the fact that the constituent services of the managed system could be operated by third parties). In the case of multiple systems, their adaptation could be still under the control of a single authority. Alternatively, it could be under the control of *multiple cooperating authorities*, that, for example, agree on some common utility objective. Finally, it could be under the control of *multiple non cooperating authorities*, that compete in a selfish way for some set of services.

2.2 The MOSES Approach to Adaptation

Devising an adaptation methodology strongly depends on the assumptions made about the domain it will be applied to. Different assumptions may lead to different formulations of the problem to be solved, and corresponding solution methodologies. Looking at the existing literature, we see that a largely uncovered region of the problem space outlined in Section 2.1 concerns the *flow of requests* granularity level. Indeed, most of the proposed methodologies focus on the *single request* case (*e.g.*, [5], [11], [16], [25], [26], [59], [61], [62]). However, a per-request approach hardly scales with workload increases, thus making this approach unsuitable for a system subject to a quite sustained flow of requests. We discuss this issue in Section 6.2.2.

With MOSES, we intend to address this part of the problem space. Indeed, MOSES focuses on a scenario where several classes of services address a relatively sustained traffic of requests to a SOA system architected as a composite service. Each class may have its own QoS requirements, and negotiates a corresponding SLA with the system. In this scenario, we assume that the QoS requirements stated in the SLA concern the *average value* of QoS attributes calculated over all the requests belonging to a *flow* generated by a given user. These values are guaranteed to the user as long as the rate of requests he/she addresses to the system does not exceed a given threshold, established in the SLA itself.

With regard to the mechanisms used to perform the adaptation, several papers have focused on *service selection*. However, it may happen that, under a specific operating condition, no selection exists of single operations offered by concrete services allowing the fulfillment of the QoS requirements. In this case, adaptation methodologies based only on service selection fail to meet their objective, which could cause a loss of income and/or reputation for a service provider.

To overcome this problem, with MOSES we propose to broaden the range of the considered adaptation mechanisms, by exploiting the availability in a SOA environment of multiple independent implementations of the same functionality. To this end, MOSES is able to select and implement adaptation actions based on a combination of both the *service selection* and *coordination pattern selection* mechanisms. In this way, MOSES may fulfill QoS levels (concerning reliability and performance) that could not be achieved otherwise, thus increasing the flexibility of a provider in facing a broader range of QoS requirements and operating conditions.

In summary, MOSES addresses the following region of the problem space characterized in Section 2.1, as evidenced by the shaded boxes in Fig. 1:

- *why*: fulfillment of SLAs about the *average value* of QoS attributes, negotiated between the provider of a composite service and multiple classes of services;

each class of services is characterized by its own SLA;
- *when*: runtime reactive adaptation;
- *where-what*:
  - composition level: services only;
  - scope (granularity): flows of requests addressed to the system by different users;
  - scope (number of systems): a single SOA system architected as a composite service;
- *how*: service selection and coordination pattern selection;
- *who*: single authority.

3 Overview of the MOSES Framework

MOSES is intended to act as a *service broker*, which offers to prospective users a composite service with a range of different service classes exploiting for this purpose a set of existing concrete services. Its main task is to drive the adaptation of the composite service to fulfill the QoS goals of the different service classes it offers, when changes occur in its operating environment.

To achieve this goal, MOSES manages a feedback control loop [45]. Figure 3 shows a high level view of the MOSES architecture implementing this loop, organized according to the IBM's MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) reference model of an autonomic system [31]. The MOSES input consists of the description of the composite service in some suitable workflow orchestration language (e.g., BPEL [49]), and the set of candidate concrete services that can be used to implement the required tasks (including the parameters of their SLAs). MOSES uses this input to build a model which is then used (and kept up to date) at runtime to determine possible adaptation actions to be performed. Each macro-component in Fig. 3 is actually architected as a set of interacting components. We give some details about these components and their functions in Section 3.4. Before that, we present in Section 3.1 the class of SOA systems managed by MOSES, in Section 3.2 the adaptation actions it performs, and in Section 3.3 a SLA model we use to state the QoS and cost requirements that drive the MOSES actions.

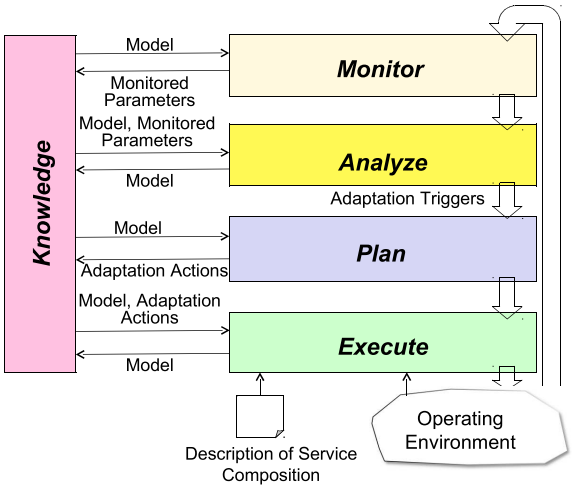


Fig. 3. The MOSES approach.

3.1 Composite Service Model

The class of services managed by MOSES consists of all those composite services whose orchestration logic (*i.e.*, their abstract composition, according to the terminology of Section 2.1) can be abstractly defined as an instance generated by the following grammar:

$$\mathrm{C} ::= \mathrm{S}|seq(\mathrm{C}^+)|loop(\mathrm{C})|sel(\mathrm{C}^+)|par\_and(\mathrm{C}^+)$$
$$\mathrm{S} ::= S_1|S_2|...|S_m$$

In this definition, C denotes a composite service, S_1, S_2, ..., S_m denote tasks (i.e., functionalities needed to compose a new added value service), and C^+ denotes a list of one or more services. Hence, MOSES currently is able to manage composite services consisting either of a single task, or of the orchestration of other services according to the composition rules: *seq*, *loop*, *sel*, *par_and*. Table 1 summarizes the intended meaning of these rules and the corresponding BPEL constructs. For the sake of clarity, in Table 2 we summarize the notation used throughout the paper.

We point out that the above grammar is purposely abstract, as it intends to succintly specify only the structure of the considered composite services. Hence, we omit details such as how to express the terminating condition for a loop. A thorough approach to the modeling of service orchestration is presented in [32], based on the *Orc* language; [20] shows how *Orc* can model the workflow patterns listed in [1]. In this respect, we point out that the grammar we define does not capture all the possible structured orchestration patterns, but includes a significant subset[1].

TABLE 1
Workflow composition rules.

Rule	Meaning	BPEL
$seq(\mathrm{C}^+)$	sequential execution of services in C^+	`sequence`
$loop(\mathrm{C})$	repeated execution of service C	`while`
$sel(\mathrm{C}^+)$	conditional selection of one service in C^+	`switch`
$par\_and(\mathrm{C}^+)$	concurrent execution of services in C^+ (with complete synchronization)	`flow`

Figure 4 shows an example of an orchestration pattern described as a UML2 activity diagram, and the corresponding instance generated by the grammar. MOSES uses this grammar to check whether the orchestration pattern of an actual SOA system matches the kind of patterns it is able to manage. In the positive case, it uses the grammar to support the construction of a suitable runtime model to be used for adaptation purposes.

1. In particular, it can be easily realized that our grammar captures the structure of workflow patterns 1, 2+3, 4, 10 (for structured cycles only), 13 and 16 reported in [1].

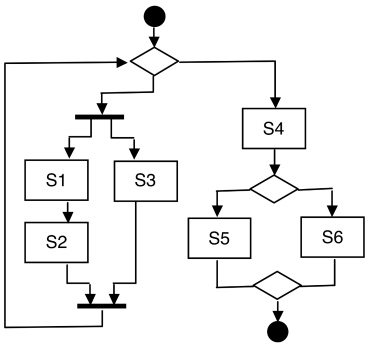


Fig. 4. A MOSES-compliant workflow.

TABLE 2
Main notation adopted in the paper.

Symbol	Description
K	Set of classes
k	Class index
$R^k_{\max}$	Class k upper bound on the expected response time
C^k	Class k cost
D^k_{min}	Class k lower bound on service reliability
λ^k_u	Class k flow of request rate generated by user u
L^k	Class k flow of request rate
S_i	Task
i	Task index
m	Number of tasks
op_{ij}	Operation/Concrete service
ij	Concrete service index
z_{ij}, $z = r\|c\|d$	Operation op_{ij} response time, cost and reliability
L_{ij}	Maximum operation op_{ij} load
$\Im_i$	Set of task i implementations
J	Implementation index
$Z(S_i; J)$, $Z = R\|\log D\|C$	Task S_i response time, cost and (log of the) reliability under implementation J
x^k_{iJ}	Fraction of class k requests for task S_i that are bound to implementation J
V^k_i	Expected number of times task S_i is invoked by a class k user
$Z^k(\boldsymbol{x})$, $Z = R\|\log D\|C$	Class k response time, cost and (log of the) reliability under adaptation policy $\boldsymbol{x}$
w_z, $z = r\|c\|d$	Normalized QoS attribute weight

3.2 Adaptation Actions

MOSES performs adaptation actions that take place at the *services only* composition level. Their goal is to determine at runtime the most suitable implementation to be bound to each abstract task S_i, selecting it from a set $\Im_i$ of available implementations, built as follows.

We assume that a set $CS = \{cs_l\}$ of candidate concrete services have been identified to build an overall implementation of the composite service. Different cs_l can be offered by different providers with different QoS and cost attributes, or even by the same provider offering differentiated services.

Each cs_l implements a set $OP(cs_l)$ of operations. We denote by $OP = \cup OP(cs_l)$ the set of all the available operations, and by $\mathcal{OP}_i \subseteq OP$ the subset of functionally equivalent operations that implement the task S_i.

MOSES exploits the availability of multiple equivalent operations to build implementations of each S_i based on the use of *redundancy* schemes, to get QoS levels possibly higher than those guaranteed by each single operation, at the expense of a higher cost. According to these schemes, a possible implementation of a task S_i may consist of a set of two or more equivalent operations belonging to $\mathcal{OP}_i$, coordinated according to some coordination pattern.

At present, the MOSES framework includes two such coordination patterns, denoted as *alt* and *par_or*, besides the simple *single* pattern. Table 3 summarizes their intended meaning. We have selected these two coordination patterns as they have complementary characteristics with respect to their QoS and cost, as will be explicitly discussed in Section 4.2.1.

TABLE 3
Coordination patterns.

Rule	Meaning
$single$	execution of a single operation
alt	sequential (alternate) execution of operations in a list, until either one of them successfully completes, or the list is exhausted
par_or	concurrent execution of the operations in a set (with 1 out of n synchronization)

Hence, the set $\Im_i$ of available implementations for each task S_i is given by the union of the following sets:

$$\Im_i = \mathcal{OP}_i \cup \mathcal{OP}^{alt}_i \cup \mathcal{OP}^{par}_i$$

where:

- $\mathcal{OP}_i$ has been already defined above; selecting an element in this set models the selection of an implementation of S_i based on a single operation;
- $\mathcal{OP}^{alt}_i$ is the set of all the ordered lists of at least two elements belonging to $\mathcal{OP}_i$, with no repetitions; selecting an element in this set models the selection of an implementation of S_i based on the *alt* pattern applied to that list;
- $\mathcal{OP}^{par}_i$ is the set of all the subsets of at least two elements belonging to $\mathcal{OP}_i$; selecting an element in this set models the selection of an implementation of S_i based on the *par_or* pattern applied to that subset.

For a given abstract composition that models the business logic of a SOA system, the selection for each S_i of different elements in the set $\Im_i$ corresponds to different concrete configurations of the overall composite service, each characterized by different values of their overall QoS attributes. We call *adaptation policy* the runtime selection and implementation of one of these configurations, to best match the QoS constraints and objectives in a

given operating environment. We detail in Sections 4 and 5 the methodology adopted in MOSES to determine this policy.

3.2.1 Adaptation Actions for Stateless and Stateful Services

In the discussion above about the MOSES adaptation actions, we implicitly assume that tasks can be bound to any concrete service implementing them. Actually, this holds only for *stateless* tasks, *i.e.*, tasks that do not require sharing any state information with other tasks. In the general case, composite services may include *stateful* tasks, *i.e.*, tasks that do need state information to be shared among them; as a consequence, these tasks need to be implemented by operations of the same concrete service. This very requirement limits the possibility of exploiting redundancy patterns to implement stateful tasks. Indeed, the functionally equivalent operations used within these patterns generally belong to different concrete services. This makes unlikely, or even impossible, the sharing of state information among them, unless we put constraints on the implementations. To overcome this problem, MOSES currently uses the *alt* or *par_or* patterns for the implementation of stateless tasks only, while the implementation of stateful tasks is restricted to only the *single* pattern.

We model the presence of *stateful* tasks by considering a partition $\mathcal{S} = \{\mathcal{S}_1, \ldots, \mathcal{S}_f\}$ of the set of tasks $\{S_1, \ldots, S_m\}$. Tasks that need to share some state information belong to the same subset $\mathcal{S}_h \in \mathcal{S}$ and need to be implemented by operations of the same concrete service cs_l. A *stateless* task S_i is simply modeled by associating it with a singleton $\mathcal{S}_h \in \mathcal{S}$.

3.3 SLA Model

As stated in Section 2.2, MOSES considers 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. MOSES presently considers the average value of the following attributes:

- *response time*: the interval of time elapsed from the service invocation to its completion;
- *reliability*: the probability that the service completes its task when invoked[2];
- *cost*: the price charged for the service invocation.

Other attributes, like reputation or availability, could be easily added.

Our general model for the SLA between the provider and the user of a service thus consists of a tuple $\langle R, C, D, L \rangle$, where: R is the upper bound on the average service response time, C is the service cost per invocation, D is the lower bound on the service reliability. The provider guarantees that thresholds R and D will hold on average provided that the request rate generated by the user does not exceed the load threshold L.

In our framework MOSES performs a two-fold role of service provider towards its users, and of service user with respect to the providers of the concrete services it uses to implement the composite service it is managing. Hence, it is involved in two types of SLAs, corresponding to these two roles, that are both defined using the SLA template. In the case of the SLAs between the composite service users and MOSES (acting the provider role), we assume that MOSES offers a set K of service classes. Hence, the SLA for user u of service class $k \in K$ is defined as a tuple $\langle R^k_{\max}, C^k, D^k_{\min}, \lambda^k_u \rangle$. All these coexisting SLAs (for each u and k) define the QoS objectives that MOSES must meet.

To meet these objectives, we assume that MOSES (acting the user role) has already identified for each task S_i a pool of concrete services implementing it. The SLA contracted between MOSES and the provider of the operation $op_{ij} \in \mathcal{OP}_i$ is defined as a tuple $\langle r_{ij}, c_{ij}, d_{ij}, L_{ij} \rangle$. These SLAs define the constraints within which MOSES should try to meet its QoS objectives.

3.4 MOSES Components

Figure 5 details the macro-components of the MOSES architecture in Fig. 3, showing the core components they consist of – *BPEL Engine*, *Composition Manager*, *Adaptation Manager*, *Optimization Engine*, *QoS Monitor*, *Execution Path Analyzer*, *WS Monitor*, *Service Manager*, *SLA Manager*, and *Data Access Library* – and their interactions. Information about their implementation is given in Section 6.

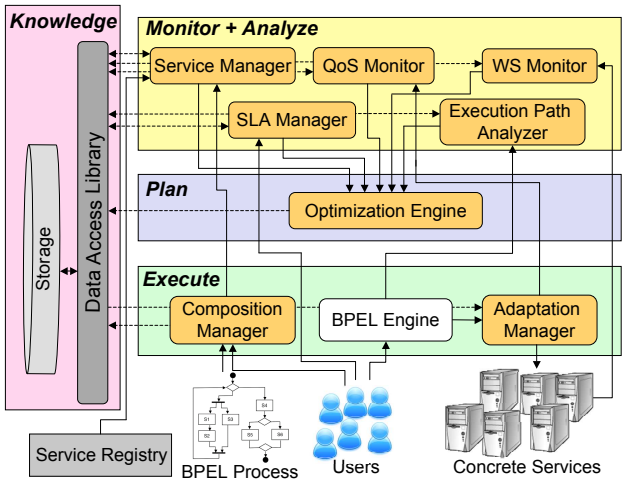


Fig. 5. MOSES high-level architecture.

The *Execute* macro-component comprises the *Composition Manager*, *BPEL Engine*, and *Adaptation Manager* components. The first component receives from the broker administrator the description of the composite service in some suitable workflow orchestration language (e.g., BPEL [49]), and builds a behavioral model of the composite service. To this end, the Composition Manager

2. This measure is called *successful execution rate* in [62].

interacts with the Service Manager for the identification of the operations that implement the tasks required by the service composition. Once created, the behavioral model is saved in the Knowledge macro-component to make it accessible to the other system components.

While the Composition Manager is invoked rarely during the MOSES operativeness, the BPEL Engine and Adaptation Manager are the core modules for the execution and runtime adaptation of the composite service. The first is the software platform that actually executes the business process (e.g., Sun BPEL Service Engine or Apache ODE) and represents the user front-end for the composite service provisioning. It interacts with the Adaptation Manager to allow the invocation of the component services. The Adaptation Manager is in charge of carrying out at runtime the adaptation actions. Indeed, for each operation invocation, it binds dynamically the request to the real endpoint that represents the operation. This endpoint is identified on the basis of the optimization problem solution determined by the Optimization Engine. We point out that the optimization problem solution takes place not for each operation invocation, but only when some component in the *Analyze* macro-component determines the need of a new solution in order to react to some change occured in the MOSES enviroment. The BPEL Engine and the Adaptation Manager also acquire raw data needed to determine respectively the usage profile of the composite service and the performance and reliability levels (specified in the SLAs) actually perceived by the users and offered by the concrete services. Together, the BPEL Engine and the Adaptation Manager are responsible for managing the user requests flow, once the user has been admitted to the system with an established SLA.

The *Optimization Engine* implements the *Plan* macro-component of the autonomic loop. It solves the optimization problem, which is based on the behavioral model initially built by the Composition Manager and instantiated with the parameters of the SLAs negotiated with both the MOSES users and the providers of the concrete services. The model is kept up to date by the monitoring activity carried out by the MOSES Monitor and Analyze macro-components. The solution of the optimization problem determines the adaptation policy in a given operating environment, which is passed to the Adaptation Manager for its actual implementation.

The components in the *Monitor* and *Analyze* macro-components capture changes in the MOSES environment and, if they are relevant, modify at runtime the behavioral model and trigger the Optimization Engine to make it calculate a new adaptation policy.

Currently, tracked changes include:

- the arrival/departure of a user with the associated SLA (SLA Manager);
- observed variations in the SLA parameters of the constituent operations (QoS Monitor);
- addition/removal of an operation implementing a task of the abstract composition (Service Manager and WS Monitor);
- variations in the usage profile of the tasks in the abstract composition (Execution Path Analyzer).

Finally, the *Knowledge* macro-component is accessed through the *Data Access Library*, which allows to access the parameters describing the composite service and its operating environment (they include the set of tasks in the abstract composition, the corresponding candidate operations with their QoS attributes, and the current solution of the optimization problem that drives the composite service implementation).

4 Adaptation and QoS Model

In this section we present the adaptation policy model adopted within MOSES, and the QoS model it uses to compute the QoS attributes of a composite service.

4.1 Adaptation Policy Model

The MOSES adaptation policy is based on a set of directives used to select at runtime the "best" implementation of the composite service in a given scenario. MOSES assumes a flow-based service demand model with multiple concurrent service classes, where for each task S_i different requests in a flow can be bound to different implementations. The MOSES adaptation policy consists of determining, for each service class k and each task S_i:

- the coordination pattern(s) and the corresponding list of operations to be used to build concrete implementation(s) for S_i (selected among the *single*, *alt* and *par_or* patterns).
- the fraction of requests generated by class k requests for S_i that must be switched and bound to a specific implementation of S_i.

We model the MOSES adaptation policy by associating with each class k a vector $\boldsymbol{x}^k = [\boldsymbol{x}_1^k, \ldots, \boldsymbol{x}_m^k]$, where each entry $\boldsymbol{x}_i^k = [x_{iJ}^k]$, $0 \leq x_{iJ}^k \leq 1$, $J \in \Im_i$, $\sum_{J \in \Im_i} x_{iJ}^k = 1$, $i = 1, \ldots, m$, denotes the adaptation policy for task S_i. Here, x_{iJ}^k denotes the fraction of class k requests for S_i to be bound to the implementation denoted by J. We denote by $\boldsymbol{x} = [\boldsymbol{x}^k]_{k \in K}$ the MOSES adaptation policy vector which encompasses the adaptation policy of all the service classes.

The adaptation policy vector $\boldsymbol{x}$ is used by the Adaptation Manager to determine for each and every invocation of a task S_i the coordination pattern to be used and the actual service(es) to implement it. Given a class k request for the task S_i, the Adaptation Manager chooses the implementation denoted by J with probability x_{iJ}^k, thus giving rise to a randomized partitioning among the implementations in $\Im_i$ of the overall class k flow directed to S_i. As an example, consider the case $\mathcal{OP}_i = \{op_{i1}, op_{i2}, op_{i3}, op_{i4}\}$ for task S_i and assume that the adaptation policy $\boldsymbol{x}_i^k$ for a given class k specifies the following values: $x_{i\{op_{i1}\}}^k = x_{i\{op_{i3}\}}^k = 0.3$, $x_{i\{op_{i2}, op_{i3}\}}^k = 0.4$ and $x_{iJ}^k = 0$ otherwise. According to this policy, given a class k request for task S_i, the Adaptation Manager binds

the request: with probability 0.3 to operation op_{i1}, with probability 0.3 to operation op_{i3}, and with probability 0.4 to the pair op_{i2}, op_{i3} coordinated by the *par_or* pattern (see Fig. 6).

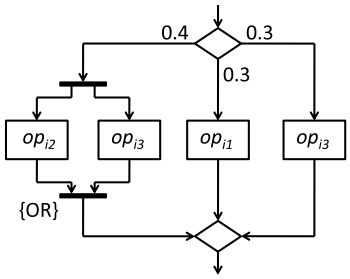


Fig. 6. Implementation of the MOSES adaptation policy for a single task.

4.2 QoS Model

MOSES presently considers the following attributes for each service class $k \in K$:

- the expected response time R^k, which is the average time needed to fulfill a class k request for the composite service;
- the expected execution cost C^k, which is the average price to be paid for a class k invocation of the composite service;
- the expected reliability D^k, which is the probability that the composite service completes its task for a class k request. As in [62], when writing expressions, we will work with the logarithm of the reliability rather than the reliability itself, to obtain linear expressions, when composing the reliability of different services.

For each service class, the overall QoS of a composite service implementation depends on: the usage profile and the composition logic of the composite service tasks; the adopted adaptation policy; the QoS of the task implementation selected within that adaptation policy.

In Section 4.2.1 we derive the QoS attributes of a task as a function of the selected implementation, while in Section 4.2.2 we show how MOSES takes into account task orchestration and usage profile to compute the composite service QoS.

QoS attributes are calculated based on the following assumptions:

- service invocation is synchronous;
- services fail according to the fail-stop model;
- service cost is charged on a per-invocation basis.

4.2.1 Task QoS Attributes

Let us first consider a task in isolation. For each class of service, the QoS of a task depends on: 1) the QoS associated with the different set of operations and the associated coordination pattern that can be bound to the task to build its concrete implementation; and 2) the probability that a particular coordination pattern and set of operations is bound to a given request.

Let $Z^k(S_i; \boldsymbol{x})$, $Z = C|D|R$, denote class k QoS attribute of task S_i under the adaptation policy $\boldsymbol{x}$. Since implementation J is chosen with probability x^k_{iJ}, we readily have:

$$C^k(S_i; \boldsymbol{x}) = \sum_{J \in \Im_i} x^k_{iJ} C(S_i; J) \tag{1}$$

$$\log D^k(S_i; \boldsymbol{x}) = \sum_{J \in \Im_i} x^k_{iJ} \log D(S_i; J) \tag{2}$$

$$R^k(S_i; \boldsymbol{x}) = \sum_{J \in \Im_i} x^k_{iJ} R(S_i; J) \tag{3}$$

where $R(S_i; J)$, $C(S_i; J)$ and $D(S_i; J)$ denote the average response time, cost and reliability of S_i, when the implementation of S_i corresponds to a given $J \in \Im_i$.

We now determine the value of these QoS attributes when S_i is implemented according to the three different coordination patterns currently considered within MOSES.

We distinguish among the three cases:

- $J \in \mathcal{OP}_i$: assuming $J = \{op_{ij}\}$, the QoS attributes coincide with those of the selected concrete operation op_{ij}:

$$C(S_i; J) = c_{ij}, \quad D(S_i; J) = d_{ij}, \quad R(S_i; J) = r_{ij} \tag{4}$$

- $J \in \mathcal{OP}^{alt}_i$: the concrete operations listed in $J = [op_{ij_1}, \ldots, op_{ij_l}]$ are tried in sequence, starting from the first in the list, until one of them successfully completes. Hence, the reliability of this pattern is derived from the probability that at least one operation completes, while the cost and time to completion of all the elements of the list must be summed, each weighted by the probability that the invocations of all the preceding elements in the list have failed:

$$\begin{aligned} C(S_i; J) &= \sum_{h=1}^{l} c_{ij_h} \prod_{s=1}^{h-1} (1 - d_{ij_s}) \\ D(S_i; J) &= 1 - \prod_{h=1}^{l} (1 - d_{ij_h}) \\ R(S_i; J) &= D(S_i; J)^{-1} \sum_{h=1}^{l} r_{ij_h} d_{ij_h} \prod_{s=1}^{h-1} (1 - d_{ij_s}) \end{aligned} \tag{5}$$

- $J \in \mathcal{OP}^{par}_i$: in this case, the costs of all the operations in $J = \{op_{ij_1}, \ldots, op_{ij_l}\}$ must be summed as they are invoked in parallel, while the completion time is the minimum of the completion times of those operations that successfully complete; thus $R(S_i; J)$ is the sum of the minimum completion time of all non-empty subsets $H \subseteq J$ weighted with the probability that only the operations in H do complete successfully:

$$
\begin{aligned}
C(S_i;J) &= \sum_{h=1}^{l} c_{ij_h} \\
D(S_i;J) &= 1 - \prod_{s=1}^{l}(1-d_{ij_s}) \\
R(S_i;J) &= D(S_i;J)^{-1} \sum_{H\in 2^J\setminus\{\emptyset\}} \Big(\prod_{j_s\in H} d_{ij_s} \prod_{j_s\in J\setminus H}(1-d_{ij_s})\Big)\cdot \\
&\quad \min_{j_s\in H}\{r_{ij_s}\}
\end{aligned}
\tag{6}
$$

We make the following remarks concerning the evaluation of $R(S_i;J)$:

- In both Equations (5)-(6) $R(S_i;J)$ is calculated conditioned on the event that at least one service in the considered list terminates. The probability of this event is equal to the service reliability $D(S_i;J)$.
- The expression for $R(S_i;J)$ in (6) is actually an approximation: the Jensen's inequality [47] ensures that the expectation of the minimum of random variables is lower than or equal to the minimum of the expectations, with the equality holding only in the deterministic case. Nevertheless, the approximation is accurate in case of small variances. In other cases a more suitable expression should be used, which would require the knowledge of the response time distribution, but this is out of the scope of this paper.

From Equations (5)-(6), we see that the implementations of S_i according to the *alt* or *par_or* patterns have the same reliability when they use the same set of services. On the other hand, it is not difficult to verify (with some algebra) that *alt* has a lower cost than *par_or*, but a higher response time, since the sequential invocation used by *alt* means that on the average not all the selected services are invoked, but the response time of those invoked must be summed.

4.2.2 QoS Attributes of the Composite Service

For each class $k \in K$ MOSES builds and maintains a labeled tree $T = (V, E, \mathcal{L})$, where V, E and $\mathcal{L}$ are the tree nodes, edges and labels, respectively. T is derived from the syntax tree that describes the production rules used to generate the composite service, by simply collapsing the S and C nodes. The leaf nodes of T are thus associated with tasks, while its internal nodes are associated with composition rules. Hence, for each non root node $v \in V$, its parent node $f(v)$ denotes the composition rule within which v occurs.

The set $\mathcal{L}$ of edges is defined as follows. Each edge $(f(v), v) \in E$ is labeled with $\ell^k(f(v), v)$, the expected number of times v is invoked within $f(v)$ for a class k request:

- if $f(v)$ is the seq or par_and composition rule then $\ell^k(f(v), v) = 1$;
- if $f(v)$ is the *loop* rule, $\ell^k(f(v), v))$ is the average number of times the loop body is executed;
- if $f(v)$ is the sel rule, $\ell^k(f(v), v)$ corresponds to the probability that v is executed.

MOSES performs a monitoring activity to keep these values up to date. Figure 7 shows the tree T maintained by MOSES for the composite service depicted in Fig. 4 (labels equal to 1 are omitted). Based on this model,

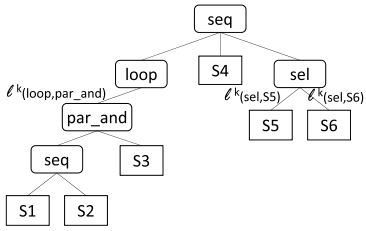


Fig. 7. Composite service labeled tree.

following well known QoS composition rules [15], we can derive the overall composite service QoS attributes $R^k(\boldsymbol{x})$, $C^k(\boldsymbol{x})$ and $D^k(\boldsymbol{x})$ (defined at the beginning of Section 4.2), given $R^k(S_i;\boldsymbol{x})$, $C^k(S_i;\boldsymbol{x})$ and $D^k(S_i;\boldsymbol{x})$, $1 \le i \le m$. Table 4 shows these rules, where for each node $v \in V$ we denote by $d(v)$ the (possibly empty) set of its children. These rules define a visit algorithm of the labeled tree T, from which we obtain:

$$Z^k(\boldsymbol{x}) = \mathtt{Z}^k(root;\boldsymbol{x})$$

$Z = C|\log D|R$, where *root* denotes the root node of T.

TABLE 4
Recursive rules to calculate the average value of the QoS attributes of a composite service.

node $v \in V$	**QoS rules** (where $\mathtt{Z}^k = \mathtt{C}^k|\log \mathtt{D}^k|\mathtt{R}^k$)
seq	$\mathtt{Z}^k(v;\boldsymbol{x}) = \sum_{u\in d(v)} \mathtt{Z}^k(u;\boldsymbol{x})$
$loop$	$\mathtt{Z}^k(v;\boldsymbol{x}) = \ell^k(v,d(v))\mathtt{Z}^k(d(v;\boldsymbol{x}))$
sel	$\mathtt{Z}^k(v;\boldsymbol{x}) = \sum_{u\in d(v)} \ell^k(v,u)\mathtt{Z}^k(u;\boldsymbol{x})$
par_and	$\mathtt{C}^k(v;\boldsymbol{x}) = \sum_{u\in d(v)} \mathtt{C}^k(u;\boldsymbol{x})$ $\log \mathtt{D}^k(v;\boldsymbol{x}) = \sum_{u\in d(v)} \log \mathtt{D}^k(u;\boldsymbol{x})$ $\mathtt{R}^k(v;\boldsymbol{x}) = \max_{u\in d(v)} \mathtt{R}^k(u;\boldsymbol{x})$
S_i	$\mathtt{Z}^k(u;\boldsymbol{x}) = Z^k(S_i;\boldsymbol{x})$

From the rules of Table 4 we now derive closed form expressions for the QoS attributes of the composite service, that will provide the basis for the optimization problem formulation of the next section. In these expressions, for each node $v \in V$, we write $v \prec u$ if node v is a descendant of node u.

Cost and Reliability. For these attributes, from the recursive rules of Table 4, it is easy to realize that

$$C^k(\boldsymbol{x}) = \sum_{i=1}^{m} \Big(\prod_{j\succeq S_i} \ell^k(f(j),j)\Big) C^k(S_i;\boldsymbol{x}) = \sum_{i=1}^{m} V_i^k C^k(S_i;\boldsymbol{x}) \tag{7}$$

and

$$\log D^k(\boldsymbol{x}) = \sum_{i=1}^{m} \Big(\prod_{j \succeq S_i} \ell^k(f(j),j) \Big) \log D^k(S_i;\boldsymbol{x}) = \sum_{i=1}^{m} V_i^k \log D^k(S_i;\boldsymbol{x}) \quad (8)$$

where $V_i^k = \prod_{l \succeq S_i} \ell^k(f(l),l)$, $S_i \in V$, is the expected number of times task S_i is invoked by the composite service for a service class k user.

Response Time. For $R^k(\boldsymbol{x})$, we need to account for the fact that the overall response time of the par_and pattern is the largest response time among its component tasks. As a consequence, the response time is no longer additive and we cannot derive an expression analogous to (7). In this case, we obtain a recursive set of expressions for the response time, whose number is linear in the number of par_and composition patterns in the process. To this end, we first introduce the notion of *direct descendant* among nodes in V. We say that a node $v \in V$ is a direct descendant of $u \in V$, denoted by $v \prec_{dd} u$, if $v \prec u$ and for any other node $w \in V$, $v \prec w \prec u$ implies $w \neq par\_and$, *i.e.*, if there is no node labelled par_and in the path from v to u. In other words, a node $v \in V$ is said to be a direct descendant of u if task/pattern v is nested within the composition pattern u, but, within u, it is not nested within a par_and pattern.

Let $\Pi \subset V$ denote the set of nodes corresponding to par_and activities. We have the following result for the response time R^k (we omit the proof - which is a simple application on the recursive formulas of Table 4 - for space reasons).

Theorem 1: For QoS class $k \in K$, the response time R^k can be computed recursively as follows:

$$R^k(\boldsymbol{x}) = \mathtt{R}^k(root;\boldsymbol{x}) \quad (9)$$

$$\mathtt{R}^k(v;\boldsymbol{x}) = \begin{cases} \max_{u \in d(v)} \mathtt{R}^k(u;\boldsymbol{x}) & v \in \Pi \\ \sum_{S_i \in V, S_i \prec_{dd} v} \frac{V_i^k}{V_v^k} R^k(S_i;\boldsymbol{x}) + & \\ \sum_{u \in \Pi, u \prec_{dd} v} \frac{V_u^k}{V_v^k} \mathtt{R}^k(u;\boldsymbol{x}) & v \notin \Pi \end{cases} \quad (10)$$

Theorem 1 provides the response time $R^k(v)$ of each composition pattern $v \in V$ and the composite service response time R^k, $k \in K$. Observe that if the par_and pattern is not present in the workflow, $\Pi = \emptyset$ and (10) reduces to $R^k(\boldsymbol{x}) = \sum_{i=1}^{m} V_i^k R^k(S_i;\boldsymbol{x})$.

5 Optimal Adaptation

In this section we present the optimization problem solved by MOSES to determine the optimal policy $\boldsymbol{x}$ in a given environment and analyze its computational complexity.

5.1 Optimization Problem

The basic goal of MOSES is to determine an adaptation policy $\boldsymbol{x}$ that allows it to meet its QoS objectives stated by the $\langle R^k_{\max}, C^k, D^k_{\min}, \lambda^k_u \rangle$ SLAs, given the constraints determined by the $\langle r_{ij}, c_{ij}, d_{ij}, L_{ij} \rangle$ SLAs. Within the possibly empty set of feasible $\boldsymbol{x}$'s that satisfy these constraints, MOSES wants to select the $\boldsymbol{x}$ that optimizes a given utility function. Depending on the utilization scenario of MOSES, the utility function could be aimed at optimizing specific QoS attributes for the different service classes (*e.g.*, minimizing their average response time) and/or it could be aimed at optimizing the MOSES own utility, *e.g.*, minimizing the overall cost to offer the composite service (that would maximize the MOSES owner incomes). These different optimization goals could be possibly conflicting, thus leading to a multi-objective optimization problem. To deal with it we transform it into a single objective problem using for this purpose the Simple Additive Weighting (SAW) technique [30], which is the most widely used scalarization method. According to SAW we define the MOSES utility function $F(\boldsymbol{x})$ as the weighted sum of the (normalized) QoS attributes of all users. More precisely, let

$$Z(\boldsymbol{x}) = \frac{\sum_{k \in K} L^k Z^k(\boldsymbol{x})}{\sum_{k \in K} L^k} \quad (11)$$

where $Z = R|\log D|C$ is the expected overall response time, reliability and cost, respectively, and $L^k = \sum_u \lambda^k_u$ is the aggregated flow of class k requests. We define the utility function as follows:

$$F(\boldsymbol{x}) = w_r \frac{R_{\max} - R(\boldsymbol{x})}{R_{\max} - R_{\min}} + w_d \frac{\log D(\boldsymbol{x}) - \log D_{\min}}{\log D_{\max} - \log D_{\min}} + w_c \frac{C_{\max} - C(\boldsymbol{x})}{C_{\max} - C_{\min}} \quad (12)$$

where w_r, w_d, $w_c \geq 0$, $w_r + w_d + w_c = 1$, are weights for the different QoS attributes. $R_{\max}$ ($R_{\min}$), $D_{\max}$ ($D_{\min}$), and $C_{\max}$ ($C_{\min}$) denote, respectively, the maximum (minimum) value for the overall expected response time, cost and reliability. We will describe how to determine these values shortly.

With these definitions, the optimization problem can be formulated as follows:

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

$$\textbf{subject to: } C^k(\boldsymbol{x}) \leq C^k, \quad k \in K \quad (13)$$

$$\log D^k(\boldsymbol{x}) \geq \log D^k_{\min}, \quad k \in K \quad (14)$$

$$\mathtt{R}^k(root;\boldsymbol{x}) + T_{ovd} \leq R^k_{\max}, \quad k \in K \quad (15)$$

$$\mathtt{R}^k(u;\boldsymbol{x}) \leq \mathtt{R}^k(v;\boldsymbol{x}), \quad u \in d(v), v \in \Pi, k \in K \quad (16)$$

$$\mathtt{R}^k(v;\boldsymbol{x}) = \sum_{S_i \prec_{dd} v} \frac{V_i^k}{V_v^k} \sum_{J \in \Im i} x^k_{iJ} R(S_i;J) + + \sum_{u \in \Pi, u \prec_{dd} v} \frac{V_u^k}{V_v^k} \mathtt{R}^k(u;\boldsymbol{x}), v \notin \Pi, k \in K \quad (17)$$

$$\sum_{k \in K} \sum_{J \in \Im i, j \in J} x^k_{iJ} V_i^k L^k \leq L_{ij}, \quad op_{ij} \in OP \quad (18)$$

$$x^k_{iJ} \geq 0, J \in \Im i, \sum_{J \in \Im i} x^k_{iJ} = 1, \quad 1 \leq i \leq m, k \in K \quad (19)$$

$$x^k_{i_1 j_1} = x^k_{i_2 j_2} \quad op_{i_1 j_1}, op_{i_2 j_2} \in OP(cs_l) \quad S_{i_1}, S_{i_2} \in \mathcal{S}_l, |\mathcal{S}_l| > 1, k \in K \quad (20)$$

Equations (13)-(17) are the QoS constraints for each class on the cost, reliability and response time. The constraints (15)-(17) for the response time are directly derived from (10). The additional term T_{ovd} accounts for the overhead introduced by the broker itself in managing the system. Equations (18) are constraints on the operations load and ensure that the system managed by MOSES does not exceed the volume of invocations agreed with the providers of those operations. The LHS of (18) is the volume of invocations of operation op_{ij} under adaptation policy $\boldsymbol{x}$. It is the sum over all service classes of the per class number of invocations per unit time of a given operation op_{ij} (the second summation is over all the implementations J in which j occurs). The RHS of (18) is the maximum load L_{ij} negotiated with the provider of the operation. Equations (19) are the functional constraints. Finally, (20) are the stateful constraints which basically require that, for stateful tasks, the fraction of requests that are bound to different operations of the same concrete service must be the same. Remember that if S_i is stateful, we only use the service selection adaptation technique; in this case J takes values only in $\mathcal{OP}_i$.

The maximum and minimum values of the QoS attributes in the objective function (12), used to get a normalized value, are determined by replacing $Z^k(x)$, $Z = R|\log D|C$ in (11) with the maximum and minimum value that the QoS attributes can attain. $R_{\max}$, $C_{\max}$, and $D_{\min}$ are simply expressed respectively in terms of $R^k_{\max}$, C^k, and $D^k_{\max}$. For example, the maximum cost is given by $C_{\max} = \frac{\sum_{k\in K} L^k C^k_{\max}}{\sum_{k\in K} L^k}$. Similar expressions hold for $R_{\max}$ and $D_{\min}$. $R_{\min}$, $C_{\min}$, and $D_{\max}$ are similarly expressed in terms of the $R^k_{\min}$, $C^k_{\min}$, and $D^k_{\max}$, the minimum response time, minimum cost and maximal reliability that can be experienced by a class k request. For instance, $C^k_{\min} = \sum_{i=1}^{m} V_i^k C^*(S_i)$ where $C^*(S_i) = min_{J\in\Im_i} C(S_i; J)$ is the minimum cost implementation of task S_i. Similar expressions hold for $R^k_{\min}$ and D^k_{max}.

We conclude by observing that the proposed optimization problem is a Linear Programming (LP) problem which can be efficiently solved via standard techniques.

5.2 Complexity Analysis

There are several algorithms to solve LP problems, including the well known simplex and interior points algorithms [37]. Widely used software packages (CPLEX®, MATLAB®) adopt variants of the well-known interior point Mehrotra's predictor-corrector primal-dual algorithm [38], which has $O(n^{\frac{3}{2}}\log\frac{n}{\epsilon})$ worst case iteration complexity and $O(n^3)$ iteration cost, where n is the number of variables of the LP problem [54]. The complexity in our problem arises from the potentially large value of n, corresponding to the number of variables x^k_{iJ}, due to the fact that J ranges over the potentially large set $\Im_i$. In the general case, we have $n = O(m|K|\max_i|\Im_i|)$. This value can grow very quickly with the number n_i of candidate operations for each task S_i.

In general, but for the simplest scenarios, we need to restrict the possible implementations to a subset of $\Im_i$. This is typically the case of sets $\mathcal{OP}_i$ of large cardinality where it is neither convenient nor feasible to consider all possible implementation patterns. To this end, we in general replace $\Im_i$ with its subset $\Im_i(a,p) = \mathcal{OP}_i \cup \mathcal{OP}_i^{alt}(a) \cup \mathcal{OP}_i^{par}(p)$, $\mathcal{OP}_i^{alt}(a) \subseteq \mathcal{OP}_i^{alt}$, $\mathcal{OP}_i^{par}(p) \subseteq \mathcal{OP}_i^{par}$ where a and p denote the maximal number of operations that can be used to implement an *alt* and *par_or* pattern, respectively. At one extreme, we have single service selection only with $\Im_i(0,0)$ where we exclude any form of redundancy; in this case, $n = O(m|K|\max_i n_i)$, which grows linearly with respect to n_i. On the other extreme, we consider all the possible redundancy coordination patterns, where the set of possible S_i implementations is $\Im(a,p) = \Im_i$. In this case, we have a superexponential number of variables $n = O(m|K|\max_i n_i!)$, since the number of possible *alt* coordination patterns of a S_i implementation is proportional to the factorial of n_i, while the number of *par_or* coordination patterns is 2^{n_i}. These values are clearly not feasible but for small values of n_i. In general, $\Im(a,p)$ with bounded a and p, limits the complexity to $n = O(m|K|(\max_i n_i)^{\max\{1,a,p\}})$. We believe that this restriction is not a significant limitation in practice, given the diminishing marginal reliability increase we can achieve with higher redundancy levels, largely outset by the increasing cost of the redundant solutions (and in case of the *alt* pattern also by increasing execution times). This theoretical analysis will be complemented by an experimental analysis in Section 6.2.

6 Experimental Results

We first describe in Section 6.1 the prototype we have developed to implement the MOSES methodology and then present the results of experiments conducted to assess its effectiveness. The purpose of this evaluation is twofold. First, we analyze in Sections 6.2 and 6.3 the performance impact of some overheads introduced by our adaptation framework. Specifically, we study the computational cost of the optimal adaptation policy carried out by the Plan macro-component, and compare it with alternative approaches in literature. Furthermore, we analyze the overhead for the runtime binding carried out by the Execute macro-component. Then, we provide in Section 6.4 an overall evaluation which involves all the MOSES macro-components to illustrate the dynamic behavior of the MOSES adaptation policy.

6.1 MOSES Prototype

The MOSES prototype has been designed following the high-level architecture shown in Fig. 5. Being a MOSES goal the capability to sustain a high traffic of requests, we have paid attention to design the prototype so as not to prejudice the performance of the managed composite services. In this section, we review the main features of the prototype; its detailed description and some preliminary experiments with a scalability focus are in [7].

The MOSES prototype exploits the rich capabilities offered by the Java Business Integration (JBI) implementation called OpenESB[3] and the relational database MySQL, which both provide interesting features to enhance the scalability and reliability of complex systems. JBI defines a messaging-based pluggable architecture and its major goal is to provide an enabling framework that facilitates the dynamic composition and deployment of loosely coupled participating applications and service-oriented integration components. The key components of the JBI environment are the Service Engines (SEs), that enable pluggable business logic, the Binding Components (BCs), that enable pluggable external connectivity, and the Normalized Message Router (NMR), which directs normalized messages from source to destination components according to specified policies. Figure 8 illustrates the OpenESB-based architecture of MOSES.

Each MOSES component is executed by one Service Engine, that can be either Sun BPEL Service Engine for the business process logic, or J2EE Engine for the logic of all the other MOSES components. The resulting prototype has a good deployment flexibility, because each component can be accessed either as standard Web service or as EJB module through the NMR. However, to increase the prototype performance, we have exploited the NMR presence for all the inter-module communications, so that message exchanges are "in-process" and avoid to pass through the network protocol stack, as it would be for SOAP-based communications. With regard to the MOSES storage layer, we have relied on the relational database MySQL, which provides transactional features through the InnoDB storage engine and supports clustering and replication. However, to free the MOSES future developers from knowing the storage layer internals, we have developed a data access library, named MOSES Data Access Library (MDAL), that completely hides the data backend. This library currently implements a specific logic for MySQL, but its interface can be enhanced with other logics.

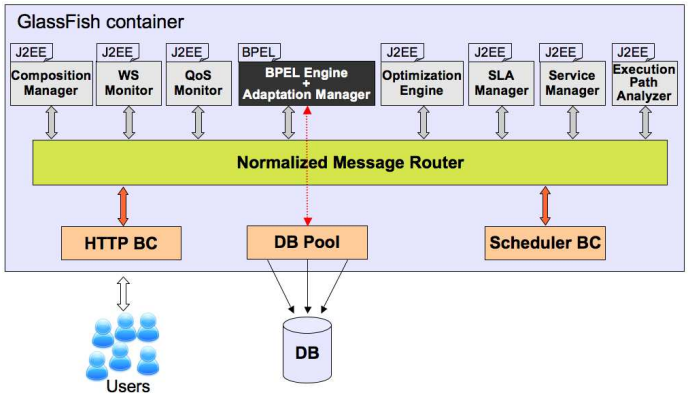


Fig. 8. MOSES OpenESB-based architecture.

3. OpenESB is a stable open source JBI implementation, developed by under the direction of Sun Microsystems.

6.1.1 MOSES Overheads

The runtime adaptation management introduces in MOSES different types of overheads, that may affect the response time of the composite service and can be classified according to the MOSES macro-components: (1) overhead due to the Plan macro-component (*i.e.*, the Optimization Engine); (2) overhead of the Execution macro-component (*i.e.*, the Adaptation Manager) due to the runtime binding of the task endpoints to concrete implementations; (3) overhead due to the Monitor and Analyze macro-components.

For the first type of overhead, we observe that the Optimization Engine calculates a new adaptation policy asynchronously with respect to the service execution flow, while incoming service requests are served by the Adaptation Manager according to the previously calculated policy. Only when the new adaptation policy is stored in the database, the Adaptation Manager begins to use it. Hence, the Optimization Engine only interferes with those requests that are being served while the new solution of the optimization problem has to be stored. However, the time taken to calculate a new adaptation policy affects the MOSES ability to promptly react to changes in the environmental conditions. Therefore, in Section 6.2 we assess the policy computational cost for increasing instances of the adaptation model and demonstrate that the optimization problem formulation as LP helps considerably in terms of load scalability with respect to other approaches in literature.

The second kind of overhead affects each request to the composite service as many times as the number of `invoke` activities executed in the BPEL process. For every invocation of an abstract task, the Adaptation Manager, which is stateless, retrieves the current adaptation policy kept in the storage layer and, according to it, determines the coordination pattern to be used and the actual operation(s) to implement the abstract task, as presented in Section 4.1. We will measure in Section 6.2 the overhead introduced by the Adaptation Manager to execute the runtime binding.

Finally, for the third kind of overhead, we should distinguish between Monitor and Analyze macro-components impact. We point out that only Monitor affects the overall service time perceived by a user, while Analyze does not affect it, since this function is executed asynchronously with respect to the business process. The most time consuming and frequent monitoring activity is that performed with respect to the SLA parameters offered by the operations. In this case, the monitoring overhead is about one millisecond for each `invoke` activity, as it only involves inserting the operation response time in a table of the MOSES database: for each operation invocation, MOSES gets the timestamp before and after the invocation itself, and then stores the observed response time, together with a flag reporting whether the operation execution failed. Such values are asynchronously read by the QoS Monitor in the Analyze

macro-component, that runs on a different machine with respect to that assigned to the BPEL execution to not interfere with the Execution macro-component. The QoS Monitor is invoked at a fixed, configurable frequency and its task is to analyze stored monitoring data in order to find out whether some SLA has been violated. It performs two steps: (i) for each invoked operation, it computes statistics like average response time and standard deviation, (ii) it compares computed statistics with SLA parameters and, in case of violation, it issues a call to the Optimization Engine.

6.1.2 Testing Environment

The testing environment consists of 3 Intel Xeon quad-core servers (2 Ghz/core) with 8 GB RAM each (nodes 1, 2, and 3), and 1 KVM virtual machine with 1 CPU and 1 GB RAM (node 4); a Gb Ethernet connects all the machines. The deployment schema of the MOSES prototype is as follows: node 1 hosted all the MOSES modules in the Execute macro-component, node 2 the data backend together with the invoked operations, and node 3 the modules in the Monitor+Analyze and Plan macro-components. Node 4 hosted the workload generator, which is different according to the experiment goal.

6.2 Adaptation Policy Computational Cost

In this section we experimentally evaluate the adaptation policy computational cost and compare it with alternative approaches in the literature.

6.2.1 Computational Cost

We implemented the optimization problem in MATLAB®. To assess the algorithm computational cost, we executed the algorithm on 2.00GHz Intel(R) Xeon(R) CPU E5504 quad-core with 8GB RAM on randomly generated problem instances and measured the solution execution time. The results are reported in Figs. 9-10 for different values of number of composite service tasks m, number of service classes $|K|$, number of operations implementing a task n_i, and different maximum degree of redundancy a, p. For the sake of simplicity, and without loss of generality, in the following we consider only the *par_or* pattern as redundancy pattern for the analysis of the computational complexity.

In Fig. 9 we plot the execution time vs the number of service tasks m for different level of *par_or* redundancy: $p = 0$, no redundancy, i.e., service selection only, $p = 2$, at most two concrete services using the *par_or* pattern, and $p = 3$, at most three concrete services using the *par_or* pattern and for different numbers of available operations implementing a given task n_i ($n_i = 10, 20$ and 50). In these set of experiments, we consider only one class of service, *i.e.*, $|K| = 1$. From the plots, we can observe that for fixed p and n_i, the execution time grows almost linearly with the number of tasks m (about one order of magnitude increase of the execution time for one order of magnitude increase in the number of tasks). At closer inspection we verified this holds true for execution times below one second; for larger values the execution time is proportional to m^3 which is consistent with the the fact that the problem size n grows linearly with m (and $|K|$) and the per iteration cost of interior points methods is $O(n^3)$. We will return to this later.

By comparing the different plots we note that, as expected, the execution time is greatly affected by the absence/presence of redundancy patterns and the number of available implementations: without redundancy (Fig. 9(a)), the execution time is always below 1 second; if we consider redundancy with the *par_or* pattern with at most two services (Fig. 9(b)), the execution time increases up to few seconds for the larger instances; by increasing the maximum number of redundant operations to three (Fig. 9(c)), the execution time grows significantly up to 5 minutes for large values of n_i. This behavior can be explained by observing that the use of the redundancy patterns, coupled with a high number of concrete operations, yields a large number of possible implementations and thus a large number of variables since n is proportional to n_i^p: in the range of values considered, while the smallest problem instance has only 100 variables, the largest one grows up to 2,087,500. This has, of course, a significant impact on the problem execution time. Nevertheless, the complexity increase caused by the exploitation of redundancy patterns should be weighted against the significant increase in reliability of the computed solution, as shown in Section 6.4.

In Fig. 10 we vary the number of service classes $|K|$ and study the impact of $|K|$ on execution time for different values of n_i and maximal redundancy level p. The number of tasks is again fixed to $m = 50$. Not surprisingly, the same remarks above on the influence of m hold true for the number of service classes: for fixed p and n_i, the execution time grows almost linearly with $|K|$ for smaller instances and proportionally to $|K|^3$ otherwise. We observe that this behaviour is consistent with the $O(n^3)$ iteration cost and $O(n^{\frac{3}{2}} \log \frac{n}{\epsilon})$ worst case iteration complexity of interior points methods. Indeed, in our experiments we observed a relatively low number of iterations for convergence, which grew only slightly from about 10 to 100 (hence much less than the $O(n^{\frac{3}{2}} \log \frac{n}{\epsilon})$, the worst iteration cost for the Mehrotra algorithm) which explains the $O(n^3)$ overall cost.

We remark that since the optimization problem is solved asynchronously with respect to MOSES operations, this large value does not directly impact on the broker responsiveness to user requests; it only affects the time it takes to update the adaptation policy. In other words, it only affects the interval of time during which, while a new solution is being computed, the broker uses the old, sub-optimal policy for the ongoing requests.

6.2.2 Comparison with Other Approaches

In this section we compare the computational complexity of the MOSES optimization problem with the complexity of other frameworks proposed in the literature for

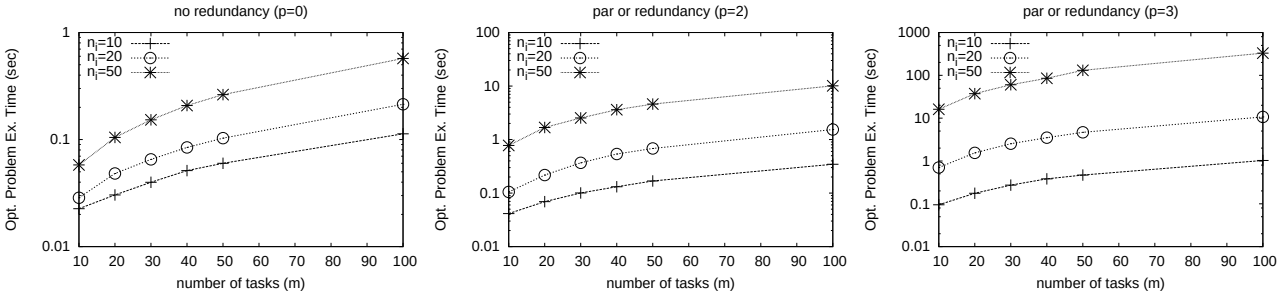


Fig. 9. Optimization problem execution time for different values of maximal redundancy: (a) no redundancy; (b) at most two concrete services using the *par_or* pattern and (c) at most three concrete services using the *par_or* pattern.

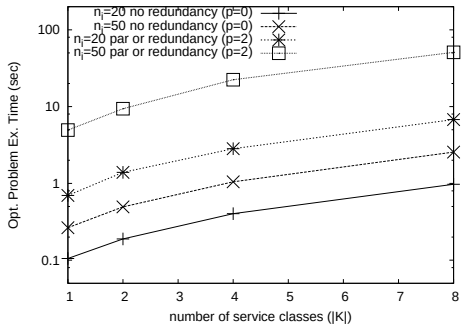


Fig. 10. Optimization problem execution time as function of the number of service classes.

runtime adaptation of SOA systems. We stress that the goal of this comparison is to show that our approach takes comparable or even smaller time to calculate an adaptation plan, and hence is at least as scalable as other approaches. As such, the following results should not be regarded as a comparison of the relative quality and/or effectiveness of the different approaches, which are not directly comparable because they differ in terms of QoS metrics, QoS model and performance goals.

For comparison we rely on published performance data. We refer to data recently published in [4] that, analogously to our approach, proposes a *per-flow* runtime adaptation framework for SOA systems. In [4], service selection takes the form of a constrained non-linear optimization problem, where non-linearities arise from: 1) the use of an explicit expression of the response time of a concrete service as function of the service load using a M/G/1 model; 2) the use of reputation - defined as the probability of not violating a threshold on the response time - as a QoS index (see [4] for details). The solution of the non-linear problem in [4] is computed through SNOPT, a commercial solver for non linear programming [23], which uses Sequential Quadratic Programming (SPT) algorithm.

For comparison purpose, we considered the same set of system parameters used in [4] (Tables 5 and 6 in [4]) and ran our experiments using randomly generated problem instances on an equivalent physical machine. Table 5 shows the average execution time in seconds of

TABLE 5
Performance comparison with the per-flow approach of [4] and per-request approaches of [5], [2] (time measured in seconds).

m	n_i	MOSES	per-flow [4]	per-request [5], [2]
100	10	0.11	8.10	0.17
100	20	0.21	9.54	0.63
100	25	0.27	9.98	0.58
100	50	0.58	14.30	0.29
1000	10	1.40	19.60	2.10
1000	20	3.03	144.30	5.38
1000	25	4.07	149.60	4.54
1000	50	8.64	451.30	19.88
5000	10	11.20	444.90	4.54
5000	20	24.55	1000.05	35.06
10000	10	15.64	970.15	113.92
10	100	0.13	7.90	0.027
10	200	0.25	9.61	0.037
10	300	0.44	9.83	0.053
10	400	0.62	10.80	0.043
10	500	0.83	13.98	0.067
10	600	0.92	15.00	0.121
10	700	1.10	17.50	0.097
10	800	1.45	17.60	0.0186
10	900	1.68	19.80	0.112
10	1000	1.87	20.50	0.170
20	500	1.78	19.30	0.189
40	500	4.47	141.40	0.432
50	500	7.54	147.30	0.560
100	500	19.22	448.70	1.518

the optimization problems in the two approaches over randomly generated problem instances. In all experiments $|K| = 1$. From the table, we can observe that the MOSES Optimization Engine is from one to two orders of magnitude faster over the large set of parameters. This directly descends from the adoption of a linear programming model as optimization problem, while in [4] service selection takes the form of a constrained non-linear optimization problem.

A direct comparison with data concerning other approaches is more problematic, as they consider *per-request* adaptation. Following [4], we compare our approach with the per-request approaches presented in [5], [2] which are among the most representative contributions in the literature. The data, also shown in Table 5, are

taken from [4] and have been obtained on an equivalent machine, according to CINT and SpecCPU2006 benchmarks (lines $(m, n_i) = (100, 10) - (10000, 10)$ report values from [5], while the rest report values from [2]). The results show that MOSES adaptation policy has execution times comparable to those in [5] and about one order of magnitude larger than those in [2]. We can argue that in a lightly loaded and/or small scale system, it may be effective to address the adaptation to each single request, independently of other concurrent requests, to customize the system with respect to that single request. However, in a large scale system subject to a quite sustained flow of requests, performing a *per-request* rather than a *per-flow* adaptation could cause an excessive computational load[4]. In this kind of scenarios, per-flow adaptation is likely to be more effective, even if it loses the potentially finer customization features of per-request adaptation. Moreover, per-request adaptation could also incur in stability and management problems, since the "local" adaptation actions could conflict with adaptation actions independently determined for other concurrent requests.

6.3 Runtime Binding

We now move on to measure the overhead introduced by the Adaptation Manager to perform the runtime binding. We point out that this kind of overhead is present in every system that provides runtime binding capabilities as MOSES does, irrespectively of the methodology used to determine the adaptation policy.

We have performed a stress test of the MOSES prototype under an open system model, where the requests to the composite service have been generated at an increasing rate through the `httperf` tool [28]. The overall experiment consists of 120 runs, each one lasting 300 seconds during which `httperf` generates requests to the composite service at a constant rate. The adaptation policy is determined at the beginning of each run and is then used for the entire duration of the run without being recalculated, because the goal of this experiment is to measure the additional overhead the runtime binding adds to a plain BPEL engine. The main performance metric we collected for each run is the mean response time, i.e., the time spent on average for the entire request-response cycle.

For increasing values of the request arrival rate to the composite service, Fig. 11 compares the response time achieved by MOSES, which executes the runtime binding according to the adaptation directives, to that obtained by the standard GlassFish ESB with Sun BPEL Engine, which only provides the composite service execution with a static binding to a given operation. As expected, MOSES is able to sustain lower load levels than GlassFish ESB before reaching the saturation point, because of the overhead introduced by the Adaptation Manager for each abstract task. Until the request arrival rate does not reach the MOSES saturation point (around 80 req/sec), the MOSES response time is on average 74% higher than that provided by GlassFish ESB (the percentage increase ranges from a minimum of 13% to a maximum of 127%). Higher request rates can be tackled by MOSES in a scalable way by replicating the system components [7]. We found that by organizing the MOSES components into clusters and replicating the clusters, we are able to minimize the network overheads for inter-module communications and storage access so that the distributed version of MOSES obtains a nearly linear performance improvement according to the number of installed GlassFish instances.

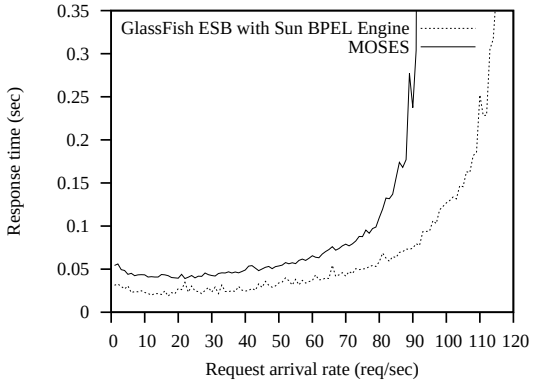


Fig. 11. MOSES response time.

In the experiments presented above, the composite service workflow corresponds to that shown in Fig. 4. In general, we observe that the runtime binding overhead is related to the size of the managed composite service. In case of static binding, the binding execution complexity depends only on the number of abstract tasks, i.e., $O(m)$. In case of MOSES runtime binding, for each invoked abstract task S_i the Adaptation Manager needs to retrieve from the database the specific records of the table that store the current adaptation policy $\boldsymbol{x}_i^k$. Since B-trees are commonly used in databases, the time complexity for searching the implementation sets is logarithmic in the number of the table entries. Therefore, the overall execution complexity in MOSES is $O(m \log(m|K| \max_i |\Im_i|))$, where the logarithmic factor is the overhead introduced by the Adaptation Manager. For space reasons, we do not report the experimental results that confirm this analysis.

6.4 MOSES-based Adaptation

We now consider all the MOSES macro-components working together and validate the effectiveness of our framework by applying it to the support of a QoS-aware composite service.

4. The Amazon e-commerce platform [21] is an example of service-based system subject to tens of millions requests. Adapting such a system according to the per-request approach would be hardly feasible.

6.4.1 Experimental Scenario

To issue requests to the composite service managed by MOSES and to mimic the behavior of users that establish SLAs before accessing the service, we have developed a workload generator. It is based on an open system model, where users requesting a given service class $k \in K$ offered by MOSES arrive at mean *user inter-arrival rate* Λ_k. Each class k user u is characterized by its SLA parameters defined in Section 3.3 and by the *contract duration* t_u^k. Each incoming user is subject to an admission control, carried out by the SLA Manager as follows. The user arrival rate λ_u^k is added to the aggregate flow L^k of class k requests currently served by MOSES, and the so obtained new instance of the optimization model is solved by the Optimization Engine. If a solution exists, the user is admitted and starts generating requests to the composite service according to the rate λ_u^k until its contract ends. Otherwise, its SLA request is rejected, because MOSES does not hold sufficient resources to manage it and the already admitted users with their SLAs, and the user terminates.

Differently from traditional Web workload, SOA workload characterization has been not deeply investigated up to now (some preliminary results can be found in [46]). Therefore, in our workload model we assume exponential distributions of parameters Λ_k and $1/t_k$ for the user inter-arrival time and contract duration, respectively. We also assume that the request inter-arrival rate and the operations service time follow a Gaussian distribution, where m_k and σ_k are the parameters of the former, and r_{ij} and $r_{ij}/12$ are the parameters of the latter.

The workload generator has been implemented in C language using the Pthreads library. Multiple independent random number streams have been used for each stochastic component of the workload model. Each experiment lasted about 5 hours and involved a minimum of 77,000 completed requests to the composite service; for each reported mean value the 95% confidence interval has been obtained with a maximum relative error in the mean value less than 0.01. The testing environment consists of 4 machines, as described in Section 6.1.2. The invoked operations hosted on node 2 are simple stubs with no real internal logic; however, their extra-functional behavior (i.e., response time, reliability, and cost) conforms to their SLA.

To illustrate the dynamic behavior of the MOSES adaptation policy, we consider again the simple abstract workflow of Fig. 4. For the sake of simplicity we assume that two candidate operations (with their respective SLAs) have been identified for each task, except for task S_2 for which four operations have been identified. The respective SLAs differ in terms of cost, reliability, and response time (being the latter measured in seconds). Table 6 summarizes the SLA parameters $\langle r_{ij}, c_{ij}, d_{ij} \rangle$ for each operation op_{ij}. They have been chosen so that for task S_i, operation op_{i1} represents the best implementation, which at a higher cost guarantees higher reliability and lower response time with respect to operation op_{ij} for $j \geq 2$, which costs less but has lower reliability and higher response time. For all operations, $L_{ij} = 10$ invocations per second.

TABLE 6
Operation SLA parameters.

Oper.	c_{ij}	d_{ij}	r_{ij}
op_{11}	6	0.995	2
op_{12}	3	0.99	4
op_{21}	4.5	0.99	1
op_{22}	4	0.99	2
op_{23}	2	0.95	4
op_{24}	1	0.95	5
op_{31}	2	0.995	1
op_{32}	1.8	0.995	2
op_{41}	1	0.995	0.5
op_{42}	0.8	0.99	1
op_{51}	2	0.99	2
op_{52}	1.4	0.95	4
op_{61}	0.5	0.99	1.8
op_{62}	0.4	0.95	4

On the user side, we assume a scenario with four classes of the composite service managed by MOSES. The SLAs negotiated by the users are characterized by a wide range of QoS requirements as listed in Table 7, with users in service class 1 having the most stringent requirements, $D_{min}^1 = 0.95$ and $R_{max}^1 = 7.1$ and users in service class 4 the least stringent requirements $D_{min}^4 = 0.85$ and $R_{max}^4 = 18.1$. The SLA cost parameters for these classes have been set accordingly, where service class 1 has the highest cost per request, $C^1 = 25$, while service class 4 only $C^4 = 12$. The rightmost column of Table 7 reports the values for L_k, that is the aggregate rate of class-k requests to the composite service. The usage profile of the different user service classes is given by the following values for the expected number of service invocations: $V_1^k = V_2^k = V_3^k = 1.5$, $V_4^k = 1$, $k \in K$; $V_5^k = 0.7$, $V_6^k = 0.3$, $k \in \{1, 3, 4\}$; $V_5^2 = V_6^2 = 0.5$. In other words, all classes have the same usage profile except for users in service class 2, who invoke the tasks S_5 and S_6 with different intensity. The values of the parameters that characterize the user workload model are $t_k = 100$ and $(m_k, \sigma_k) = (3, 1)$, $\forall k \in K$.

TABLE 7
Class SLA parameters.

Class k	C^k	$D^k_{\min}$	$R^k_{\max}$	L^k
1	25	0.95	7.1	1.5
2	18	0.9	11.1	1
3	15	0.9	15.1	3
4	12	0.85	18.1	1

We have estimated the MOSES overhead for each served request, represented by T_{ovd} in Equation 15, to be around 100 msec in the testing environment used for the experiments. This overhead includes 50 msec due to the Adaptation Manager and the BPEL process execution (see Fig. 11, when the request arrival rate varies between 2 and 12 req/sec due to the considered setting of our workload parameters), and 50 msec for the begin/commit transaction overhead due to MySQL. For the experiments presented in the next section, the changes detected by MOSES and that trigger the Opti-

mization Engine include only the arrival/departure of users, that cause a variation of the load and QoS requirements addressed to the composite service. We recall that the MOSES prototype is able to capture a variety of changes in its environment (listed in Section 3.4) and to trigger consequently the Optimization Engine for a new adaptation policy. For space reasons, in the experimental results we consider only one type of adaptation events. Nevertheless, due to the setting of our workload parameters, the corresponding mean adaptation rate is on average 0.02 req/sec (corresponding to the mean interarrival rate of new contract requests), that is the solution of a new instance of the optimization problem is on average calculated every 1.2 minutes.

6.4.2 Runtime Adaptation Results

We illustrate the result of the adaptation directives issued by MOSES under two different scenarios of the broker goal: 1) the maximization of the average reliability, *i.e.*, w_d=1; 2) the minimization of the average cost, *i.e.*, w_c=1.

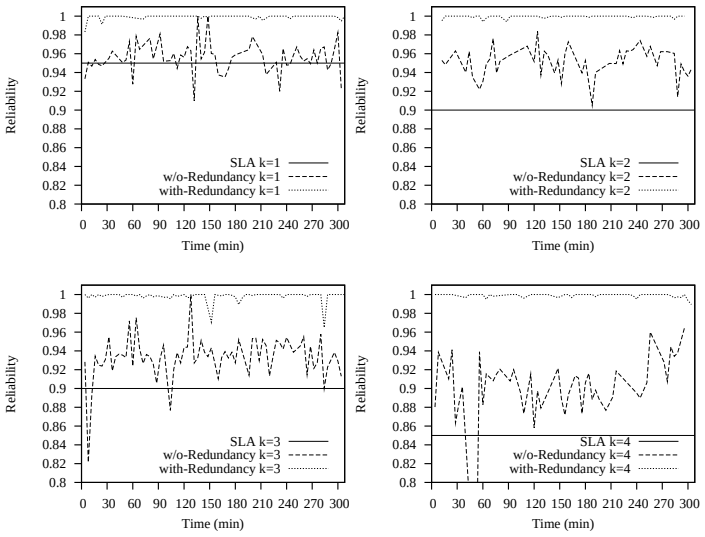
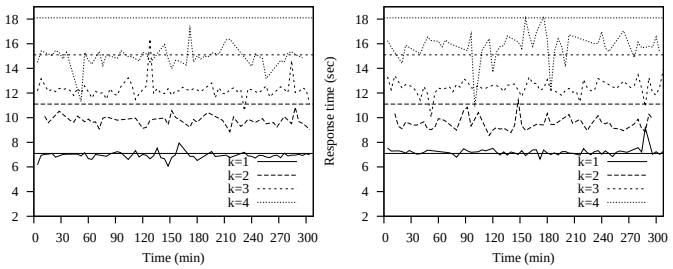


Fig. 12. Scenario 1 ($w_d = 1$): reliability over time.

In both sets of experiments, we analyze the effectiveness of considering redundancy patterns for the tasks implementation. To this end, we compare the performance of a broker that supports all the three patterns (*par_or*, *alt*, and *single*) with that of a broker that supports only the *single* pattern. In the first case (denoted by **with-Redundancy**), the formulation of the optimization problem is in Section 5; in the latter case (denoted by **w/o-Redundancy**), we solved the same optimization problem with $\Im_i$ replaced by $\Im_i(0,0)$. The results are summarized in Table 8, which shows for each class the measured values of the SLA parameters for the with- and w/o-Redundancy approaches in the two scenarios, along with the 95% confidence interval.

In the first scenario, the broker goal is to maximize the users' reliability. In this setting, the solution provided by



(a) w/o-Redundancy. (b) with-Redundancy.

Fig. 13. Scenario 1 ($w_d = 1$): response time over time.

the Optimization Engine is bounded by the maximum cost the broker is willing to pay for each user (which defines its profit margin). Only for the w/o-Redundancy approach, the solution is also bounded by the single operations available to implement the services. Both approaches succeed in respecting the SLA values (see left side of Table 8). We observe that with respect to the w/o-Redundancy approach, the with-Redundancy approach allows achieving a higher level of satisfaction of the reliability parameter (the mean values for the four classes range from 0.9983 to 0.9991) at a higher cost, whose mean value is saturated to the maximum agreed in the SLA (see Table 7). This is particularly evident for class 1, which requires the most stringent performance requirements at the highest cost (the mean cost ranges from 21.149 for the w/o-Redundancy approach to 25.051 for the with-Redundancy approach, being 25 the cost settled in the SLA). The improvement of the reliability is achieved thanks to the additional patterns *par_or* and *alt* exploited by the with-Redundancy approach.

To compare in more detail the w/o- and with-Redundancy approaches with respect to the reliability QoS parameter, Fig. 12 shows how in the first scenario the reliability of the composite service varies over time for the four classes. The horizontal line is the agreed reliability, as reported in Table 7. We observe that the w/o-Redundancy approach leads to some violations of the agreed reliability, while the with-Redundancy approach allows the broker to offer always a reliability much better than that agreed.

The exploitation of the redundancy coordination patterns improves the reliability but it can determine an increase in the response time when the *alt* pattern is selected. Figure 13 shows how in the first scenario the response time of the composite service varies over time for the four classes, being the horizontal lines the agreed response times, as reported in Table 7. We observe that the with-Redundancy approach leads to a response time that is slightly higher than that achieved by the w/o-Redundancy approach. However, for classes from 2 to 4 the response time is always much lower than that agreed, while for class 1, which requires the most stringent performance requirements, it reaches the maximum agreed in the SLA.

TABLE 8
Measured values for SLA parameters (mean and 95% confidence interval).

	Scenario 1 (w_d=1) - w/o-Redundancy			Scenario 2 (w_c=1) - w/o-Redundancy		
	C^k	D^k	R^k	C^k	D^k	R^k
$k=1$	21.149 ± 0.148	0.955 ± 0.0028	6.934 ± 0.037	20.973 ± 0.172	0.9539 ± 0.0033	7.007 ± 0.044
$k=2$	18.173 ± 0.155	0.9514 ± 0.0035	9.741 ± 0.075	15.866 ± 0.117	0.934 ± 0.0036	10.899 ± 0.079
$k=3$	14.808 ± 0.072	0.9339 ± 0.0024	12.194 ± 0.058	12.255 ± 0.062	0.9032 ± 0.003	14.491 ± 0.076
$k=4$	11.744 ± 0.093	0.9017 ± 0.0049	14.936 ± 0.122	10.659 ± 0.09	0.8623 ± 0.0058	17.651 ± 0.135
	Scenario 1 (w_d=1) - with-Redundancy			**Scenario 2 (w_c=1) - with-Redundancy**		
	C^k	D^k	R^k	C^k	D^k	R^k
$k=1$	25.051 ± 0.184	0.9991 ± 0.0004	7.182 ± 0.045	20.843 ± 0.172	0.9555 ± 0.0033	7.135 ± 0.051
$k=2$	18.427 ± 0.137	0.9991 ± 0.0004	9.509 ± 0.068	15.891 ± 0.141	0.9308 ± 0.0044	11.023 ± 0.1
$k=3$	14.97 ± 0.074	0.9987 ± 0.0003	12.641 ± 0.064	12.144 ± 0.053	0.9024 ± 0.0026	14.747 ± 0.066
$k=4$	11.953 ± 0.087	0.9983 ± 0.0004	16.001 ± 0.121	10.426 ± 0.091	0.8625 ± 0.0062	17.76 ± 0.146

We now turn our attention to the second scenario, where 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 users more than the minimum required. As a result, the solution provided by the Optimization Engine guarantees only the minimum required level of reliability (see right side of Table 8), with increasing costs for increasing reliability levels.

Let us now consider how in the second scenario the reliability of the composite service varies over time, as shown in Fig. 14. As expected, we find that the reliability level achieved with the with-Redundancy approach is lower with respect to the first scenario. The motivation is that, when the broker minimizes the cost of the composite service, the solution of the optimization problem exploits less frequently the redundancy coordination patterns *par_or* and *alt* as they may cost more than the *single* pattern.

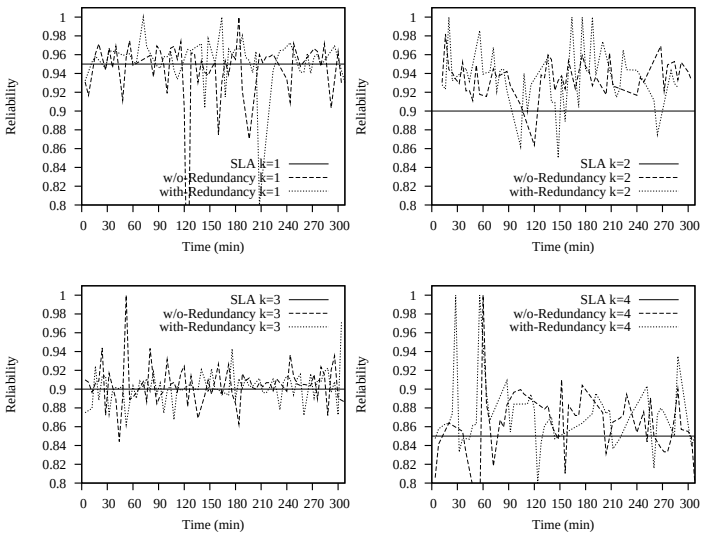


Fig. 14. Scenario 2 ($w_c = 1$): reliability over time.

7 Related Work

7.1 Architectures for Self-adaptation

It has been widely recognized that the architecture of self-adaptive software systems should include one or more control loops to perform self-adaptation tasks [17]. A notable example of a general approach based on this idea is the *autonomic computing* framework [29]. As evidenced in Section 3, MOSES can be seen as an instantiation for the SOA environment of an autonomic system, focused on the fulfillment of QoS requirements.

A reference model for the architecture of a self-adaptive software system has been presented in [33]. This paper suggests to architect the system along three different layers, that interact with each other by reporting status information to the above layer and issuing adaptation directives to the layer below. The bottom layer (*component control*) is concerned with adaptation at the level of single components (*i.e.*, services in the SOA domain). The middle layer (*change management*) reactively uses a pre-specified set of plans to adapt the system consisting of components at the lower layer. When these plans are no longer able to meet the system goals, or when new goals are introduced, the upper layer (*goal management*) determines new adaptation plans.

From the viewpoint of this three-layer reference model, the bottom layer of the MOSES framework includes the set of concrete services used in the service composition, plus the QoS Monitor, Service Manager, and WS Monitor components. Indeed, each concrete service possibly implements its own adaptation actions to fulfill the QoS goals it has negotiated. The QoS Monitor, Service Manager, and WS Monitor components collect and report to the above middle layer status information (reliability, delivered QoS) about these services. The middle layer of MOSES includes those components (Adaptation Manager and Optimization Engine) that use the status information from the layer below to determine a new adaptation policy to be used for the composite service implementation. In the MOSES framework, this layer bases its actions on a pre-defined set of candidate concrete services and a given utility function to be optimized. Both can be changed by the upper goal

management layer, by modifying the information stored in the MOSES Knowledge component.

Finally, the whole system consisting of MOSES itself plus the composite service it manages can be considered as a single concrete service offered to prospective users, thus appearing to those users as a bottom layer component with self-adaptation capabilities, that can be used as a basic building block of a larger self-adaptive system.

7.2 Methodologies for QoS Driven Adaptation

According to the characterization of the problem space given in Section 2.1, we discuss here how the different questions have been addressed by the existing literature, evidencing also some uncovered issues.

Why. Most of the existing approaches addressing the fulfillment of QoS requirements concern the average values of QoS attributes. Specifically, some approaches deal with a single quality attribute (e.g., response time in [35], reliability in [22], [56], and cost in [12]), while others are able to tackle multiple quality attributes defining simple aggregate QoS functions (e.g., [5], [11], [42], [39], [62]). A potential limitation of these approaches lies in the fact that user perceived QoS is often better expressed in terms of bounds on the percentile of the QoS metrics, as also reflected in some commercial practices[5]. To the best of our knowledge, only the approaches proposed in [24], [60] offer guarantees on the percentile of the response time. The results in [60], though, are limited to sequential patterns and only apply to the *single* request scenario, while [24] proposes a heuristic for request scheduling in a single database server which is based on the prediction of execution time.

A related basic problem to be solved when dealing with requirements about QoS attributes of SOA systems is how to determine their value for a composite service, given the QoS delivered by its component services. Some papers have focused on this specific issue [15], [35], [53], while others deal with it as a step within the more general problem of QoS based model-driven runtime adaptation of SOA systems. MOSES currently deals with requirements concerning the average value of multiple QoS attributes.

When. Existing approaches can be placed between the link/load time and run time stages [5], [11], [42], [39], [62], [41], as expected in the SOA domain. These approaches basically adopt a reactive mode to deal with adaptation. A topic that deserves more investigation concerns proactive adaptation. A paper considering this issue is [27]. MOSES currently adopts a reactive mode.

Where-What. Some works consider both *services and workflow* as the overall *composition level* where adaptation takes place. For example, the SOA environment redundancy is exploited in [16], [26], [59] to identify multiple diverse workflows that can be used under different operating conditions to achieve the same goal. A different approach, called SASSY and proposed in [41], generates service-oriented architectures based on quality requirements. Based on an initial model of the required service types and their communication, SASSY generates an optimal architecture by selecting the best services and potentially adding patterns such as replication or load balancing, so allowing also some kind of workflow restructuring. However, most of the proposed methodologies address the problem working at the *services only* composition level, using different mechanisms to determine the concrete implementation to be bound to each workflow task (as discussed below for the *how* question).

From the viewpoint of the considered *number of systems* and *granularity level* at which adaptation is performed (adaptation *scope* dimension), most of the proposed approaches focus on a scenario concerning a *single system* and a *single request* addressed to that system, as already pointed out in Section 2.2 ([5], [11], [16], [25], [26], [40], [59], [61], [62]). Given this common reference scenario, these papers propose different methodologies to determine the most suitable adaptation action. Some of them propose heuristics (e.g., [8], [25] or genetic algorithms in [11]) to determine the adaptation actions. Others propose exact algorithms to this end: Yu and Lin [61] formulate a multi-dimension multi-choice 0-1 knapsack problem as well as a multi-constraint optimal path problem; Zeng et al. [62] present a global planning approach to select an optimal execution plan by means of integer programming; in [5], [25], [52] the adaptation actions are selected through mixed integer programming; while [40] combines optimization techniques and heuristic approaches.

MOSES too refers to a single SOA system to be managed, but focuses on *per-flow*, rather than on *per-request* adaptation, and determines the adaptation actions to be performed by solving a linear programming model.

How. Several papers have focused on dynamic *service selection*, such as [5], [11], [42], [39], [40], [41], [62]. Others have instead considered the dynamic *coordination pattern selection*. For example, [25] provides a methodology to select different redundancy schemes to improve the reliability experienced by a single request addressed to a composite service. [58] proposes a flexible heuristic provisioning strategy that allocates multiple services for unreliable tasks in order to proactively deal with failures. Finally, [34] presents an example of adaptation based on *service tuning*, using for this purpose a management interface implemented according to the WSDM standard.

In this respect, the MOSES aim is to provide a unified framework where service selection is integrated with coordination pattern selection, to achieve a greater flexibility in the adaptation of a SOA system.

Who. Existing approaches mainly focus on systems managed by a single authority. At present MOSES is defined for a single system managed by a single authority, but the approach can be extended in a quite straightforward way to multiple services managed by

5. The Amazon SOA-based e-commerce platform [21] includes SLAs concerning the 99.9 percentile of the response time under a given peak load of service requests per second.

single authority, and to multiple services managed by multiple cooperating authorities. Runtime adaptation issues in a scenario where multiple services managed by multiple non cooperating authorities compete for shared resource are instead, to the best of our knowledge, still largely unexplored, even if this scenario seems to be quite likely for the SOA domain.

Finally, an important aspect in model-driven adaptation of SOA systems concerns the assumptions underlying the proposed methodologies. In this respect, even if not always explicitly stated, most of the proposed approaches share a common set of assumptions. In particular, they include: (i) synchronous invocation of services, and (ii) stateless services. The former assumption is relevant for the estimation of the overall response time as (possibly weighted) sum of the response time of the invoked operations. The latter provides the ground to freely (re-)bind different functionally equivalent operations to an abstract task, and to coordinate them by redundancy patterns. A relaxation of the stateless assumption can be found in [5], where the proposed model allows to specify that different operations belonging to the same concrete services must be bound to corresponding abstract tasks with an "all or none" logic.

MOSES too relies on the synchronous invocation assumption to calculate the overall response time. On the other hand, MOSES is able to deal with both stateless and stateful tasks, but it limits the use of the *alt* or *par_or* patterns to stateless tasks, as described in Section 3.2.1.

8 Lessons Learned and Conclusions

In this paper we presented the MOSES framework for runtime QoS-driven adaptation of SOA systems. The basic guideline we have followed in its definition has been to devise an adaptation methodology that is *flexible*, to cope with QoS requirements that may come from different classes of users, and (as much as possible) *efficient*, to make it suitable for runtime operations. To achieve flexibility, we have presented a novel approach which allows us to integrate within our framework different adaptation mechanisms (service selection and coordination pattern selection) that can be simultaneously used to serve the requests of different users, or even different requests from the same user. Our results show that, actually, including both these mechanisms in the MOSES toolset allows coping with a broader range of dependability requirements. To achieve efficiency, we have considered a per-flow granularity which also allowed us to formulate the optimal adaptation problem as an LP problem. Our experiments have indeed shown that our approach has comparable or less computational cost than alternative approaches in the literature. Nevertheless, the inclusion of redundancy patterns can result in excessive computational costs given the large number of alternative implementations to consider for larger problem instances. This suggests to limit the use of these patterns for a subset of the tasks (e.g., the most critical ones) or to scenarios where the achievement of a higher dependability is mandatory.

Because of the distributed nature of the SOA environment, the QoS perceived by a user of the composite service can be affected by the performance of the networking infrastructure used to access the selected component services. In the current version of MOSES this aspect is not explicitly included. A possible way to manage within the MOSES framework the impact of networking services on the overall user perceived QoS could be to include these services in the workflow that specifies the service composition. This implies that suitable SLAs should be negotiated and monitored with the involvement of network providers (as discussed for example in [57]), and taken into account when determining the optimal service selection.

We have presented a fully functional prototype which implements the MOSES framework. The prototype is presently based on a centralized architecture implementing the whole MAPE control loop, as outlined in Section 3, which may suffer from scalability issues. To cope with them, a possible approach is to architect MOSES as a decentralized system consisting of a set of federated MOSES brokers, with each one of them exploiting partially overlapping sets of concrete services. In this architecture, the brokers coordinate themselves according to a *master-slave* scheme, where slave brokers actually implement only the Monitor and Execute functions of the MAPE loop. The whole loop is implemented by the master broker, that receives monitored data from slaves, and uses them to build and solve an overall optimization problem (through its Optimization Engine module), that combines together the respective goals and constraints. The calculated adaptation policy is then transmitted to slave brokers that implement it through their respective Adaptation Manager modules. On the positive side, this master-slave architecture can be easily implemented, with only minor modifications, from the current centralized implementation. Indeed, we have already implemented it on a local scale, as pointed out in Section 6.3. On the negative side, the master broker could still represent a bottleneck. Moreover, it appears suitable for a single organization offering QoS-aware adaptive services, that need to cope with scalability issues caused by high volumes of requests. It could be less suitable in the case of multiple organizations.

A more scalable and decentralized solution would consist in distributing the whole MAPE loop among multiple MOSES brokers. Under the hypothesis of federated cooperating brokers, this would require to devise a distributed solution of the overall optimization problem. With respect to the current implementation, this would require a change in the Optimization Engine algorithm and implementation. Under the hypothesis of competing brokers, MOSES should be more deeply restructured. In this respect, we note that our characterization of the problem space of self-adaptation for SOA systems evidences that the case of several self-adaptive SOA

systems under cooperating or non-cooperating scenarios is not yet satisfactorily covered by current literature. Hence, investigating how to cope with these issues is a timely and promising indication for our future work on the MOSES framework.

Besides this, there are several other directions along which we plan to continue our work on the MOSES framework, as we outline below. A first direction consists in dealing with requirements concerning higher moments and percentiles of QoS attributes. In this respect, a first step towards the inclusion of percentile-based SLAs in MOSES is presented in [13]. Moreover, we are investigating how to extend the set of assumptions under which MOSES currently works. This includes: relaxing the synchronous invocation assumption; considering alternative failure models (e.g., Byzantine failures, which require different kinds of redundancy patterns); including additional orchestration patterns for service composition, with respect to those matching the grammar presented in Section 3. A further direction is related with the assumption, in the current MOSES framework implementation, of a known pool of candidate concrete services, without considering how this pool can be selected and possibly changed at runtime, and the relevant SLA parameters dynamically negotiated. This is a relevant issue, and dealing with it should be one of the tasks of the upper layer of MOSES, according to the three-layers model presented in [33].

Acknowledgements

The authors would like to thank the Editor and the anonymous referees for their helpful comments that definitely contribute to the improvement of the paper.

This work has been partially supported by the European Community's Seventh Framework Programme under project Q-ImPrESS and IDEAS-ERC Project SMScom.

References

[1] W. Aalst, A. Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[2] M. Alrifai and T. Risse. Combining global optimization with local selection for efficient qos-aware service composition. In *Proc. WWW '09*, pages 881–890, 2009.

[3] J. Andersson, R. de Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 27–47. Springer, 2009.

[4] D. Ardagna and R. Mirandola. Per-flow optimal service selection for web services based processes. *J. Syst. Softw.*, 83(8), 2010.

[5] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *IEEE Trans. Softw. Eng.*, 33(6):369–384, 2007.

[6] L. Baresi, E. Di Nitto, and C. Ghezzi. Toward open-world software: Issue and challenges. *IEEE Computer*, 39(10):36–43, 2006.

[7] A. Bellucci, V. Cardellini, V. Di Valerio, and S. Iannucci. A scalable and highly available brokering service for SLA-based composite services. In *Proc. ICSOC '10*, volume 6470 of *LNCS*, pages 527–541. Springer, Dec. 2010.

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

[9] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web Services Architecture (WSA). W3C Working Group Note 11 Feb. 2004.

[10] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel. Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17(5):309–332, 2005.

[11] G. Canfora, M. Di Penta, R. Esposito, and M. Villani. A framework for QoS-aware binding and re-binding of composite web services. *J. Syst. Softw.*, 81(10):1754–1769, 2008.

[12] L. Cao, J. Cao, and M. Li. Genetic algorithm utilized in cost-reduction driven web service selection. In *Proc. CIS '05*, volume 3802 of *LNCS*, pages 679–686. Springer, 2005.

[13] V. Cardellini, E. Casalicchio, V. Grassi, and F. Lo Presti. Adaptive management of composite services under percentile-based service level agreements. In *Proc. ICSOC '10*, volume 6470 of *LNCS*, pages 381–395. Springer, Dec. 2010.

[14] V. Cardellini, E. Casalicchio, V. Grassi, F. Lo Presti, and R. Mirandola. QoS-driven runtime adaptation of service oriented architectures. In *Proc. ACM ESEC/SIGSOFT FSE*, 2009.

[15] J. Cardoso. Complexity analysis of bpel web processes. *Software Process: Improvement and Practice*, 12(1):35–49, 2007.

[16] G. Chafle, P. Doshi, J. Harney, S. Mittal, and B. Srivastava. Improved adaptation of web service compositions using value of changed information. In *Proc. IEEE ICWS '07*, 2007.

[17] B. H. C. Cheng, H. Giese, P. Inverardi, J. Magee, and R. de Lemos. 08031 – software engineering for self-adaptive systems: A research road map. In *Software Engineering for Self-Adaptive Systems*, volume 08031 of *Dagstuhl Seminar Proceedings*. IBFI, 2008.

[18] M. Colombo, E. D. Nitto, and M. Mauri. Scene: A service composition execution environment supporting dynamic changes disciplined through rules. In *Proc. ICSOC '06*, volume 4294 of *LNCS*, pages 191–202. Springer, 2006.

[19] M. Colombo, E. D. Nitto, M. D. Penta, D. Distante, and M. Zuccalà. Speaking a common language: A conceptual model for describing service-oriented systems. In *Proc. ICSOC '05*, volume 3826 of *LNCS*, pages 48–60. Springer, 2005.

[20] W. R. Cook, S. Patwardhan, and J. Misra. Workflow patterns in Orc. In *Proc. COORDINATION '06*, volume 4038 of *LNCS*, pages 82–96. Springer, 2006.

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[22] S. Gallotti, C. Ghezzi, R. Mirandola, and G. Tamburrelli. Quality prediction of service compositions through probabilistic model checking. In *Proc. QoSA '08*, volume 5281 of *LNCS*, pages 119–134. Springer, 2008.

[23] P. Gill, W. Murray, and M. Saunders. SNOPT: an SQP algorithm for large-scale constrained optimization. *SIAM J. on Optimization*, 12(4):979–1006, 2002.

[24] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper. Adaptive quality of service management for enterprise services. *ACM Trans. Web*, 2(1):1–46, 2008.

[25] H. Guo, J. Huai, H. Li, T. Deng, Y. Li, and Z. Du. Angel: Optimal configuration for high available service composition. In *Proc. IEEE ICWS '07*, pages 280–287, 2007.

[26] J. Harney and P. Doshi. Speeding up adaptation of web service compositions using expiration times. In *Proc. WWW '07*, 2007.

[27] J. Hielscher, R. Kazhamiakin, A. Metzger, and M. Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In *Proc. ServiceWave '08*. Springer, 2008.

[28] httperf. http://www.hpl.hp.com/research/linux/httperf/.

[29] M. C. Huebscher and J. A. McCann. A survey of autonomic computing - degrees, models, and applications. *ACM Comput. Surv.*, 40(3), 2008.

[30] C. Hwang and K. Yoon. *Multiple Criteria Decision Making, Lecture Notes in Economics and Mathematical Systems*. Springer, 1981.

[31] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[32] D. Kitchin, A. Quark, W. R. Cook, and J. Misra. The Orc programming language. In *Proc. FMOODS/FORTE '09*, volume 5522 of *LNCS*, pages 1–25. Springer, 2009.

[33] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. *Proc. IEEE FOSE '07*, pages 259–268, 2007.

[34] P. Martin, W. Powley, K. Wilson, W. Tian, T. Xu, and J. Zebedee. The wsdm of autonomic computing: experiences in implementing autonomic web services. In *Proc. SEAMS '07*, 2007.

[35] M. Marzolla and R. Mirandola. Performance prediction of web service workflows. In *Proc. QoSA '07*, volume 4880 of *LNCS*, pages 127–144. Springer, 2007.

[36] P. McKinley, S. Sadjadi, E. Kasten, and B. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, 2004.

[37] N. Megiddo. On the complexity of linear programming. In *Advances in Economic Theory: 5th World Congress*, pages 225–268. T. Bewley, ed., Cambridge University Press, Cambridge, 1987.

[38] S. Mehrotra. On the implementation of a (primal-dual) interior point method. *SIAM J. on Optimization*, 2:575–601, 1992.

[39] D. A. Menascé. Qos issues in web services. *IEEE Internet Comp.*, 6(6):72–75, 2002.

[40] D. A. Menascé, E. Casalicchio, and V. Dubey. A heuristic approach to optimal service selection in service oriented architectures. In *Proc. WOSP '08*, pages 13–24. ACM, 2008.

[41] D. A. Menascé, J. M. Ewing, H. Gomaa, S. Malek, and J. P. Sousa. A framework for utility-based service oriented design in sassy. In *Proc. WOSP/SIPEW '10*, pages 27–36. ACM, 2010.

[42] D. A. Menascé, H. Ruan, and H. Gomma. QoS management in service oriented architectures. *Perform. Eval.*, 7-8(64), 2007.

[43] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *Proc. WWW '08*, pages 815–824. ACM, 2008.

[44] H. R. Motahari-Nezhad, J. Li, B. Stephenson, S. Graupner, and S. Singhal. Solution reuse for service composition and integration. In *Proc. WSCA '09*, 2009.

[45] H. Müller, M. Pezzè, and M. Shaw. Visibility of control in adaptive systems. In *Proc. ULSSIS '08*, pages 23–26. ACM, 2008.

[46] P. Nagpurkar, W. Horn, U. Gopalakrishnan, N. Dubey, J. Jann, and P. Pattnaik. Workload characterization of selected jee-based web 2.0 applications. In *Proc. IEEE IISWC '08*, pages 109–118, 2008.

[47] R. Nelson. *Probability, stochastic processes, and queueing theory*. Springer-Verlag, New York, 1995.

[48] E. D. Nitto, C. Ghezzi, A. Metzger, M. P. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.*, 15(3-4):313–341, 2008.

[49] OASIS. Web Services Business Process Execution Language Version 2.0, Jan. 2007.

[50] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[51] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Proc. IEEE WISE '03*, 2003.

[52] Y. Qu, C. Lin, Y. Wang, and Z. Shan. Qos-aware composite service selection in grids. In *Proc. IEEE GCC '06*, pages 458–465, 2006.

[53] D. Rud, A. Schmietendorf, and R. Dumke. Performance modeling of ws-bpel-based web service compositions. In *Proc. IEEE Services Computing Workshops*, pages 140–147, 2006.

[54] M. Salahi and T. Terlaky. Mehrotra-type predictor-corrector algorithm revisited. *Optimization Methods Software*, 23(2), 2008.

[55] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.

[56] N. Sato and K. S. Trivedi. Stochastic modeling of composite web services for closed-form analysis of their performance and reliability bottlenecks. In *Proc. ICSOC '07*. Springer, 2007.

[57] J. Skene, F. Raimondi, and W. Emmerich. Service-level agreements for electronic services. *IEEE Trans. Softw. Eng.*, 36(2):288–304, 2010.

[58] S. Stein, T. R. Payne, and N. R. Jennings. Flexible provisioning of web service workflows. *ACM Trans. Internet Technol.*, 9(1), 2009.

[59] K. Verma, P. Doshi, K. Gomadam, J. A. Miller, and A. P. Sheth. Optimal adaptation in web processes with coordination constraints. In *Proc. IEEE ICWS '06*, pages 257–264, 2006.

[60] K. Xiong and H. Perros. Sla-based service composition in enterprise computing. In *Proc. IEEE IWQoS '08*, 2008.

[61] T. Yu, Y. Zhang, and K.-J. Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Trans. Web*, 1(1):1–26, 2007.

[62] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnamam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5), May 2004.