MOSES: a Platform for Experimenting with QoS-driven Self-adaptation Policies for Service Oriented Systems

Valeria Cardellini[1], Emiliano Casalicchio[2⋆], Vincenzo Grassi[1], Stefano Iannucci[1], Francesco Lo Presti[1], Raffaela Mirandola[3]

[1] Università di Roma "Tor Vergata", 00133 Roma, Italy
Dip. di Ingegneria Civile e Ingegneria Informatica
cardellini@ing.uniroma2.it, vincenzo.grassi@uniroma2.it,
iannucci@ing.uniroma2.it, lopresti@info.uniroma2.it
[2] Blekinge Institute of Technology, 37141 Karlskrona, Sweden
Dept. of Computer Science and Engineering
emiliano.casalicchio@bth.se
[3] Politecnico di Milano, 20133 Milano, Italy
Dip. di Elettronica, Informazione e Bioingegneria
raffaela.mirandola@polimi.it

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 chapter we present MOSES, a software platform supporting QoS-driven adaptation of service-oriented systems. It has been conceived for service-oriented systems architected as composite services that receive requests generated by different classes of 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. Besides providing its own self-adaptation functionalities, MOSES lends itself to the experimentation of alternative approaches to QoS-driven adaptation of service-oriented systems thanks to its modular architecture.

1 Introduction

The *service-oriented architecture* (SOA) paradigm encourages the realization of new software systems by composing network-accessible loosely-coupled services. Given their intrinsic characteristics, SOA-based systems are characterized by a continuous evolution: providers may modify the exported services; new services may become available; existing services may be discontinued by their providers;

⋆ Accepted for publication in *Software Engineering for Self-Adaptive Systems: Assurances*, R. de Lemos, D. Garlan, C. Ghezzi, and H. Giese (eds.), Lecture Notes in Computer Science Vol. 9640, Springer, to appear.

⋆ Work done when the author was affiliated with Università di Roma "Tor Vergata".

usage profiles may change over time; variations in the communication infrastructure may impact the reachability of existing services [7]. As a consequence, SOA-based systems represent a typical domain where the introduction of self-adaptation features can give significant advantages in fulfilling and maintaining over time a given set of non-functional requirements concerning the delivered quality of service (QoS).

Within this framework, we present in this chapter MOSES (*MOdel-based SElf-adaptation of SOA systems*), a software platform for QoS-driven runtime self-adaptation of service oriented systems. MOSES is tailored for a utilization scenario where a SOA system architected as a composite service needs to fulfill either single requests or a sustained traffic of requests generated in both cases by several classes of users. Within this scenario, MOSES determines the most suitable system configuration for a given operating environment by solving an optimization 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 obtaining QoS levels that could not be achieved by using single services. In this respect, MOSES is a step forward with respect to several proposed approaches for runtime SOA systems adaptation, which limit the range of their actions to the selection of single services to be used in the composition.

MOSES is architected as a centralized broker, which advertises and offers to prospective users the composite service it manages. To fulfill the functional and non-functional requirements agreed on with the service users, MOSES implements the functionalities envisaged in the MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) reference model for autonomic systems [20], thus acting as a runtime controller for the composite service. Special care has been given in implementing these functionalities according to a modular architecture, so that different implementations can be easily plugged into the overall MOSES framework. This makes MOSES amenable for the experimentation of different adaptation policies aimed at maintaining the QoS delivered by a composite service. As shown in the following sections, this has allowed us to experiment with adaptation policies tailored for different utilization scenarios (e.g., with more or less stringent QoS requirements, or different models of service demand), and can allow other researchers to experiment with their own policies. Therefore, MOSES can be of help to get confidence about the ability of a given adaptation policy in providing guarantees that the system QoS requirements are satisfied. In this respect, in Sec. 2.3 we briefly discuss MOSES in terms of the benchmarking criteria proposed in Chap. XXX of this book (*Perpetual Assurances in Self-Adaptive Systems*), to assess the MOSES capabilities in providing assurances for SOA systems and in Sec. 5 we compare MOSES with existing approaches according to these criteria.

The main features and the overall design of MOSES have been already presented in [12]. In this chapter, we provide the detailed design of the MOSES

platform and present some insights of its implementation, with the goal of providing a framework that can be used and modified by the self-adaptive systems software engineering community. To this end, we release the source code of MOSES, which is freely available with an opensource license at `http://www.ce.uniroma2.it/moses`. We hope that this software can provide a platform for developing and experimenting with different QoS-driven adaptation mechanisms exploited in the context of service oriented systems. On the web site, along with the source code, we also provide the documentation of the MOSES modules, so that the interested researchers and practitioners can either modify the existing mechanisms or plug their own into MOSES.

The remaining of this chapter is organized as follows. In Sec. 2 we classify MOSES within a frame of reference for QoS-driven self-adaptation of SOA systems and characterize its adaptation capabilities with respect to a case study that has been proposed as a testbed for the benchmarking criteria proposed in this book (see Chap. Chap. XXX of this book (*Perpetual Assurances in Self-Adaptive Systems*)). In Sec. 3 we outline the MOSES architecture and the main tasks of its components. In Sec. 4 we dive into the MOSES implementation. In Sec. 5 we review related work, focusing on frameworks similar to MOSES, and discuss their adaptation capabilities according to the benchmarking criteria proposed in this book. Finally, we conclude in Sec. 6, summarizing some lessons learned with the MOSES development and presenting directions for future work.

2 MOSES Framework

2.1 Problem Space Characterization

To better delineate the contribution given by MOSES, we briefly outline a characterization of the problem space of QoS-driven self-adaptation for the SOA domain, taken from [12], providing a frame of reference for MOSES itself and for other contributions in the existing literature. Figure 1 summarizes the main concepts of this characterization, which are briefly described in the following. We refer to [12] for a thorough discussion of these concepts. The taxonomy is built around some basic questions and challenges for the whole domain of self-adaptive software systems [36], and the corresponding possible answers based on the specific features of the SOA domain, with special emphasis on QoS aspects:

- *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.

Why. Non functional requirements concerning the delivered QoS and cost, are usually expressed in the SOA domain by *Service Level Agreements* (SLA) [27]. In a stochastic setting, a SLA can concern guarantees about the *average value* of quality attributes, or about the *higher moments* or *percentiles* of these attributes.

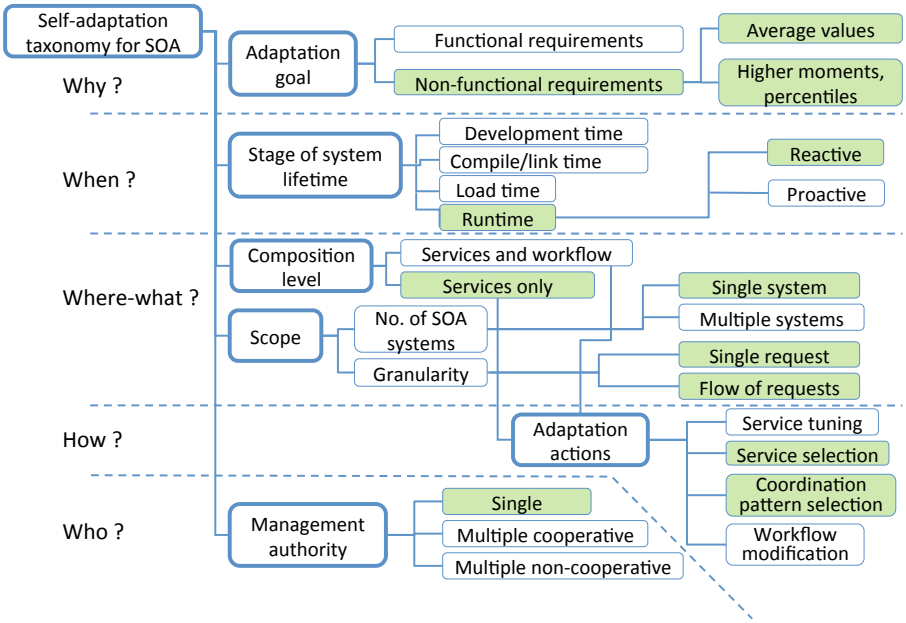


Fig. 1: Taxonomy of self-adaptation for SOA.

When. In the SOA domain, adaptation usually occurs at runtime, and can be *reactive* or *proactive*. In reactive approaches, collected data are evaluated and the adaptation to problematic events takes place after they have occurred. Proactive approaches take advantages of predictive models to detect in advance the need of changes, thus possibly invoking the adaptation before the SLA violation could actually happen.

Where-What. From the viewpoint of *service composition*, adaptation in the SOA domain may take place at two different levels: (*i*) *concrete services only*: the adaptation only involves the concrete composition, acting on the implementation each abstract task is bound to, leaving unchanged the composition logic (*i.e.*, the overall abstract composition); (*ii*) *services and workflow*: the adaptation involves both the concrete and abstract composition; in particular, the composition logic can be altered.

From the viewpoint of its *scope*, adaptation in the SOA domain can be characterized in terms of: (*i*) the *granularity* level at which adaptation is performed; (*ii*) the *number* of SOA systems operating in the same environment that are directly involved in the adaptation process. In the first case (granularity), adaptation can concern the fulfilment of requirements for a *single service request*, or the overall *flow of requests* addressed to a SOA system. In the second case (number of SOA systems), adaptation can focus on a *single SOA system*, or *multiple systems* competing for overlapping sets of services.

How. Possible adaptation actions include: (*i*) *service tuning*; (*ii*) *service selection* (selection for each abstract task of a matching *single* operation offered by a concrete service); (*iii*) *coordination pattern selection* (selection for each abstract task of a *set* of functionally equivalent operations offered by different concrete services, coordinating them according to some spatial or temporal redundancy pattern, e.g., k-out-of-n); (*iv*) *workflow modification* (modification or reconfiguration of the workflow composition logic, e.g., through the activation and deactivation of features in the variability model [2]).

Who. In case of multiple SOA systems, their adaptation can be under the control of: (*i*) a *single authority*; (*ii*) *multiple cooperating authorities*; (*iii*) *multiple non cooperating authorities*.

2.2 System Model

MOSES manages the runtime self-adaptation of composite services. The overall utilization scenario where MOSES is intended to operate can be outlined as follows (we refer to [12] for a more detailed description).

Managed system. MOSES manages the class of SOA systems consisting of composite services whose orchestration logic can be described by a structured workflow built using the following composition rules: (*i*) *sequential* composition; (*ii*) *conditional selection*; (*iii*) *conditional iteration*; (*iv*) *fork-join* parallel composition.

Service demand. MOSES focuses on a scenario where several classes of users address their requests to a composite service. Each class may have its own QoS requirements, and negotiates a corresponding SLA with the system. The current MOSES implementation includes modules that allow the management of SLAs concerning either each *single* request, or the overall *flows* of requests generated by different users.

Environment dynamics. The MOSES goal is to keep the managed system able to fulfil the QoS requirements of its users despite the occurrence of variations in the system operating environment. Currently, tracked variations that could trigger an adaptation action include: (*i*) the arrival/departure of a user with the associated QoS requirements; (*ii*) observed variations in the QoS attributes of the concrete services used by the service composition; (*iii*) addition/removal of concrete services implementing some task of the abstract composition; (*iv*) variations in the usage profile of the tasks in the abstract composition.

Adaptation actions. MOSES dynamically binds each abstract service specified in the composite service workflow to a concrete implementation taken from

a known set of available services. When the current bindings result no longer adequate to fulfil the existing QoS requirements (as a consequence of an occurred environment variation) the adaptation action performed by MOSES consists in a change of some of those bindings. Given an abstract service S in the workflow, possible changes currently include: (i) binding S to an implementation consisting of a single concrete service; (ii) binding S to an implementation consisting of a set of functionally equivalent services, coordinated according to a sequential *try until success* pattern (greater reliability and higher cost with respect to single service binding); (iii) binding S to an implementation consisting of a set of n functionally equivalent services, coordinated according to a parallel *1-out-of-n* pattern (smaller response time and higher cost with respect to single service binding).

We point out that actually the MOSES adaptation actions include not only the dynamic deterministic binding of an abstract service to a concrete implementation, but also the dynamic probabilistic binding, where the implementation is probabilistically selected from a suitable set of alternatives. This allows, for example, the probabilistic partitioning of requests in a flow among different implementations, thus giving more flexibility in achieving the required QoS level. More details can be found in [12, 13].

QoS requirements. MOSES offers to the prospective concurrent users of the composite service it manages the possibility of specifying QoS requirements expressed as min/max thresholds on the average value or percentile of some QoS attributes. In the current MOSES implementation, considered QoS attributes include the service *response time*, *cost* and *reliability*.

Besides trying to fulfil the threshold-based QoS requirements of the composite service users, MOSES can also take into consideration an additional requirement concerning the optimization of a function of the composite service QoS attributes. Depending on the utilization scenario, this (utility) function could be aimed at optimizing specific QoS attributes for different classes of users (*e.g.*, minimizing their experienced 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. MOSES deals with it by using the Simple Additive Weighting (SAW) technique [18].

To fulfil these requirements, MOSES relies on the availability of a set of concrete services that can be used to instantiate the abstract services in the managed composite service. Each concrete service in this set is assumed to offer a SLA concerning the guarantees it gives about its QoS attributes, provided that the load generated by the user does not exceed a given threshold (specified in the same SLA). The guarantees specified in the SLA are analogous to those managed by MOSES, i.e., concerning min/max thresholds on the average value or percentile of the QoS attributes.

To manage the resulting overall set of QoS requirements and constraints, and to determine the suitable adaptation actions, MOSES instantiates and solves a suitable instance of the following optimization problem template:

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

where $\boldsymbol{x}$ is the decision vector for the adaptation actions to be performed, $F(\boldsymbol{x})$ is the objective function, $Q^{\alpha}(\boldsymbol{x})$ and $Q^{\beta}(\boldsymbol{x})$ are, respectively, those QoS attributes for which a max or min threshold is specified, $S(\boldsymbol{x})$ are the constraints on the offered load, and $\boldsymbol{x} \in A$ is a set of functional constraints on the $\boldsymbol{x}$ value.

Within the MOSES framework, variations in the QoS requirements (e.g., new QoS thresholds specified by the users, or new utility objectives to be achieved) can be simply managed by a suitable re-instantiation of the optimization problem to be solved.

Problem space region covered by MOSES. In summary, the colored boxes in Fig. 1 evidence the regions of the problem space resulting from the characterization outlined above that are covered by MOSES.

2.3 Benchmarking Criteria

Chap. XXX of this book (*Perpetual Assurances in Self-Adaptive Systems*)proposes some benchmarking criteria that can be used to compare different approaches for "perpetual assurances for self-adaptive systems". The same chapter also presents in Sec. 2.3 a case study (Tele Assistance System (TAS)) that specifically refers to the SOA domain considered by MOSES. The TAS case study goal is to provide a set of specific challenges for that domain, which can be used as a testbed for the capabilities of a self-adaptation approach to fulfil requirements driven by the proposed benchmark criteria.

We defer to Sec. 5 a discussion of the compliance of MOSES with some of the benchmarking criteria. In the same section we also compare MOSES with other existing approaches according to the same criteria. In this section, instead, we briefly discuss how MOSES can tackle the challenges discussed for the TAS scenario (we refer to each challenge with the name used for it in Sec. 2.3 of Chap. XXX of this book (*Perpetual Assurances in Self-Adaptive Systems*), and refer to that chapter for its detailed description).

S1 - Individual service failure. Since concrete services implementing the Alarm Service may have different failure rates and costs, MOSES is able to select the single concrete service (or some redundancy pattern) that fulfils the cost and reliability requirements. In particular, MOSES tackles this challenge by solving

the following optimization problem received as input:

$$\begin{aligned} &\textbf{min } cost(AlarmService) \\ &\textbf{subject to: } \lambda(AlarmService) \leq X \end{aligned}$$

where $\lambda(s)$ denotes the failure rate of service s.

S2 - Variation of failure rate of services over time. MOSES deals with variation at runtime of the cost and/or failure rate of different concrete services for the Medical Analysis Service and Alarm Service thanks to its continuous monitoring activity, which can lead to a recalculation of a new solution for the optimization problem provided as input. In particular, analogously to case **S1**, this corresponds in the MOSES framework to solving a different instance of the optimization problem:

$$\begin{aligned} &\textbf{min } cost(AlarmService) + cost(MedicalAnalysisService) \\ &\textbf{subject to: } \lambda(AlarmService) + \lambda(MedicalAnalysisService) \leq X \end{aligned}$$

S3 - New service becomes available. Assuming that the newly arrived instance of Alarm Service is added to the pool of available services considered by MOSES, this addition is an event that triggers the recalculation of the optimization problem solution (which could possibly lead to selecting the newly arrived Alarm Service instance).

S4 - New type of service becomes available. Adding a new service type to an already existing workflow implies a replanning of the overall execution plan that was managed by the original workflow. Presently, MOSES is not designed to automatically deal with this kind of scenario. The new workflow must be built outside of the MOSES framework, and then provided to MOSES as a new input.

3 MOSES High-level Architecture

Figure 2 shows the MOSES architecture, whose modules are organized according to the MAPE-K loop – *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. A discussion about their implementation is presented in Sec. 4.

The *Execute* macro-component comprises the *Composition Manager*, *BPEL Engine*, and *Adaptation Manager* modules. The first module receives from the broker administrator the description of the composite service in some suitable workflow orchestration language (e.g., BPEL [33]), and builds a behavioral model of the composite service. To this end, the Composition Manager interacts with the Service Manager for the identification of the operations that implement the tasks required by the service composition. Once created, the system model is saved in the MOSES storage (i.e., the Knowledge macro-component) to make it accessible to the other system modules.

While the Composition Manager is invoked rarely during the MOSES operativeness, the BPEL Engine and Adaptation Manager are the core modules for the

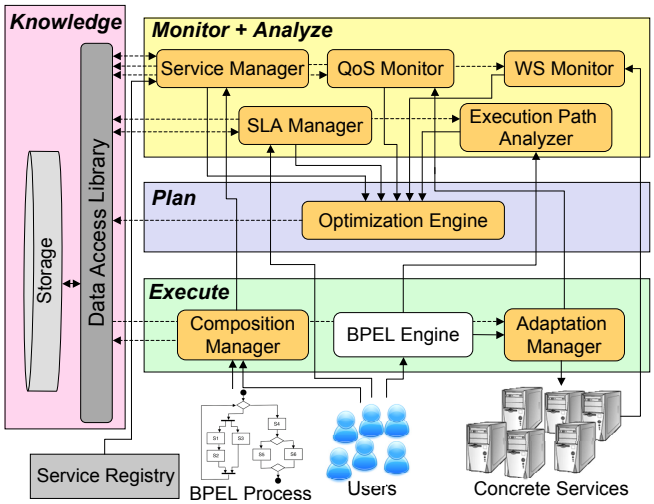


Fig. 2: MOSES high-level architecture.

execution and runtime adaptation of the composite service. The BPEL Engine is the software platform that actually executes the composite service, described in BPEL [33], 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 the actuator at runtime of the adaptation actions. Indeed, for each operation invocation, it binds dynamically the request to the real endpoint(s) that represents the operation. This(these) endpoint(s) is(are) identified on the basis of the optimization problem solution determined by the Optimization Engine. 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.

The *Optimization Engine* implements the *Plan* macro-component of the MAPE-K 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 saved on the MOSES Storage and retrieved by the Adaptation Manager for its actual implementation.

The modules in the *Monitor* and *Analyze* macro-components capture changes in the MOSES environment and, if they are relevant, modify at runtime the system model kept in the Storage layer and trigger the Optimization Engine to make

it calculate a new adaptation policy. For each adaptation trigger mentioned in Sec. 2.2, we indicate here the corresponding MOSES module(s) responsible for tracking it: (*i*) the arrival/departure of a user with the associated SLA, possibly performing an admission control [1] (SLA Manager); (*ii*) observed variations in the SLA parameters of the constituent concrete services (QoS Monitor); (*iii*) addition/removal of an operation implementing a task of the abstract composition, the latter due either to graceful failures or crashes (Service Manager and WS Monitor); (*iv*) variations in the usage profile of the abstract tasks in the service composition (Execution Path Analyzer).

Finally, the *Knowledge* macro-component is accessed through the MOSES *Data Access Library* (MDAL), 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 MOSES Prototype

We have designed the MOSES prototype on the basis of the high-level architecture presented in the previous section and shown in Fig. 2. In this section, we first review the main features of the software prototype, which has been presented in [12, 8]; then, in Sec. 4.1 we present the relevant details regarding the implementation of the core MOSES modules and in Sec. 4.2 we discuss how MOSES can be extended to support a new service selection policy that may require the monitoring of additional QoS parameters. In Sec. 4.3 we discuss the overheads introduced by the adaptation policies and mechanisms implemented by MOSES. Finally, in Sec. 4.4 we briefly describe the evaluation tool we developed to test the MOSES performance and which is available within the MOSES package.

The MOSES prototype exploits the rich capabilities offered by the OpenESB framework [34] and the relational database MySQL, which both provide interesting features to enhance the scalability and reliability of complex systems. OpenESB, which was initially designed and developed under the direction of Sun Microsystems and is currently maintained by its own community, is a Java-based open source Enterprise Service Bus (ESB) that meets the requirements of SOA and provides a stable and lightweight JBI implementation. JBI, which stands for Java Business Integration, is considered as a philosophy for system integration, describing how to define and use a virtual bus to communicate between components. More precisely, it 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 JBI specification defines as core components the Service Engines (SEs), the Binding Components (BCs), and the Normalized Message Router (NMR). The SEs enable pluggable business logic and receive messages from the bus and send messages to the bus; the BCs enable pluggable external connectivity, being able to generate bus messages upon

receipt of stimuli from an external source, or to generate an external action in response to a message received from the bus; the NMR directs normalized messages from source to destination components according to specified policies. Figure 3 illustrates the OpenESB-based architecture of MOSES.

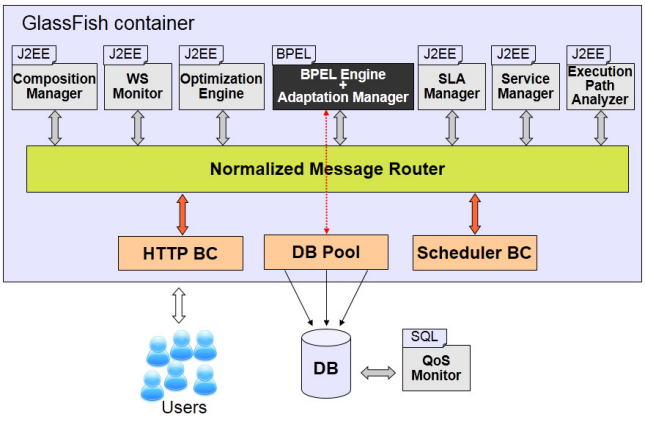


Fig. 3: MOSES OpenESB-based architecture.

Each MOSES component is executed by one SE, that can be either Sun BPEL Service Engine (a highly scalable orchestrator based on BPEL 2.0) for the business process logic, or J2EE Engine for the logic of the other MOSES modules. 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 rely on the relational database MySQL. 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.

4.1 MOSES Modules

We now describe the implementation of the core MOSES modules for driving the adaptation decisions (Optimization Engine), executing at runtime the adaptation decisions (Adaptation Manager), monitoring the QoS attributes of the concrete services offered by the third-party service providers (QoS Monitor and WS Monitor), managing the arrival and departure of users with the associated

SLA (SLA Manager), and storing the knowledge needed by the MAPE-K control loop (storage layer). Since the remaining modules (Composition Manager, Service Manager, Execution Path Analyzer) that offer some useful but supplementary functionalities are not yet fully developed, they are currently not included in the MOSES distribution.

Optimization Engine The Optimization Engine is the MOSES module that computes the solution of the optimization problem sketched in Sec. 2.2. Currently, MOSES supports two classes of optimization strategies for service selection (*per-flow* and *per-request*), corresponding to two different granularity levels.

At the per-request grain, the adaptation focuses on each single request submitted to the system, e.g., [3, 4, 10, 11, 25]; it aims at fulfilling the QoS constraints of that specific request, thus allowing potentially finer customization features. However, most per-request policies exhibit scalability and stability problems in a large scale system subject to a quite sustained flow of requests, because each request is managed independently of all the other concurrent ones [13].

The per-flow grain considers the flow of requests of a user rather than the single request, and the adaptation goal is to fulfill the QoS constraints that concern the global properties of that flow, e.g., the average composite service response time or its availability, e.g., [6, 12, 21]. However, adaptation policies adopting the per-flow grain are not able to ensure strict fulfillment of the required QoS attributes to each single request.

In MOSES we implemented three alternative optimization policies:

- the per-flow policy we presented in [12], where the service selection and the coordination pattern selection are jointly addressed with an efficient Linear Programming (LP) formulation;
- the per-request policy proposed by Ardagna and Pernici [4];
- the load-aware per-request we proposed in [13]; it relies on a Mixed Integer Linear Problem (MILP) formulation and exploits the multiple available implementations of each abstract task, thus realizing a runtime probabilistic binding that allows to achieve a randomized load balancing among the different concrete services available for the same functionality.

The Optimization Engine module is not self-contained as it relies on some external optimization software package to solve the optimization problem. In the current implementation, we use MATLAB® [24] for the per-flow class and CPLEX® [19] for the per-request class. In both optimization problem classes, the solution providing the current adaptation policy is stored into MOSES's MySQL database through MDAL.

Adaptation Manager The Adaptation Manager, which is responsible of the runtime binding of the abstract tasks to the corresponding implementations, is a proxy module interposed between the BPEL process and the concrete services. It is called every time the BPEL Engine needs to invoke an external Web service. Differently from the other MOSES modules, it is not implemented as a JBI

Service Engine. It is rather implemented as a standard Java class belonging to the application server classpath, and thus it is accessible by any application served by the application server itself. Its implementation offers a generic interface to the BPEL process so that it can be invoked with any kind of SOAP message as payload, wrapped in a proper envelope. Its task is to extract the SOAP message from the envelope and to drive it to the correct concrete service(s) according to the information received through the headers of the envelope and a predefined policy. To this end, the Adaptation Manager reads the most up-to-date adaptation policy plan from the database using the MDAL library, invokes the concrete service(s), and forwards the service response to the BPEL Engine.

We observe that the BPEL process needs to be customized in order to interact with the Adaptation Manager of MOSES. Such changes are all about replacing the invocation to concrete services with invocations to the Adaptation Manager and can be simply accomplished by following the instructions we provide with the MOSES documentation.

QoS Monitor The QoS Monitor in MOSES employs a passive methodology to collect the actual values of QoS attributes provided by the concrete services, in particular response time and reliability. Data are collected without injecting additional load, but rather observing the system behavior in response to actual service invocations. Indeed, at each concrete service invocation, the perceived QoS is stored on a continuous time basis in the database using the MDAL library. Rather than using active probes that generate additional traffic on the monitored concrete services, we exploit passive collectors so to reduce the monitoring costs. However, this may result in a slower detection of SLA violations and can thus be appropriate when timeliness is not critical for the system [15].

The QoS Monitor is actually implemented as a MySQL stored procedure for performance reasons, being activated every time the Adaptation Manager invokes a concrete service. In the current MOSES implementation, we consider only the service response time as monitored QoS parameter; however, the extension to consider other QoS parameters is trivial.

To analyze the collected data regarding the response time, we use the online adaptive cumulative sum (Cusum) algorithm [30] for service response time monitoring and abrupt change detection. As discussed in [13], the online adaptive Cusum detector we implemented combines an Exponential Weighted Moving Average (EWMA) filter to track the slow varying response time series average with a two-sided Cusum test to detect abrupt changes in the series average which cannot be timely accounted by EWMA filter.

The Analyze phase of the MAPE-K control loop is completed by a detached thread of the Optimization Engine. Whenever the QoS Monitor detects that a SLA violation in the response time of some concrete service, it sets to true a flag stored in the database. Such a flag is periodically checked by the Optimization Engine detached thread: if it is set to true, the Optimization Engine calculates the new adaptation policy using the updated model, where the perceived QoS values measured by the QoS Monitor replace those agreed in the SLA.

WS Monitor The WS Monitor, which is implemented as a Web service and executed by a J2EE Service Engine, is configured to periodically probe all the concrete services known to MOSES in order to find out which services are currently available. Whenever the WS Monitor finds that some service changed its state (going from running to failed or vice-versa), it sends a trigger to the Optimization Engine. In its turn, the latter, using the updated model that reflects the service availability/unavailability, computes the new adaptation policy, that will be then executed by the Adaptation Manager.

SLA Manager The SLA Manager performs a coarse-grain admission control in MOSES. Under the per-flow adaptation policy, it allows MOSES to decide whether to accept or reject a new potential user in such a way to guarantee non-functional QoS requirements to its already admitted users. Such admission control mechanism is required because the candidate concrete services, with whom the broker has defined a SLA, can be in a limited number with respect to the incoming request load and can thus be unable to provide the QoS levels required by the prospective users, as well as those of ongoing users already in the SOA system.

In the current MOSES implementation, we provide only a myopic admission control strategy, where the broker takes admission decisions using only the present system state, on the basis of the SLA of the requesting user and the SLAs of its currently admitted users. To this end, the SLA Manager invokes the Optimization Engine adding the requirements of the new potential user; if the instance of the per-flow optimization problem can be solved, then the contract will be established, otherwise it will be discarded. In [1] we proposed a forward-looking admission control policy based on Markov Decision Processes (MDP), with the goal to maximize the broker discounted reward while guaranteeing non-functional QoS requirements to its users. However, the MDP-based policy is not yet implemented into the MOSES prototype.

The SLA Manager allows also to free up resources when an existing contract expires; to this end, it invokes again the Optimization Engine to determine a new adaptation policy.

Storage Layer The storage system behind MOSES is accessed through the MOSES Data Access Library (MDAL). Such module is of fundamental importance since it holds all the information needed to optimally drive the adaptation; therefore, it has been designed and implemented so to not constitute a bottleneck for the entire system. The goal of MDAL is to decouple the stored data from the programming interfaces used to access it: as a result, MOSES developers do not need to know how data are actually stored because they are hidden behind a simple interface. Furthermore, the internals of the storage layer could be changed in the future without affecting other MOSES modules. Specifically, MDAL provides a set of entities that model the domain in which MOSES operates and a set of implemented interfaces to read, write, update, and delete the entities.

For the storage, MOSES relies on a relational database. Its ER diagram, shown in Fig. 4, includes as main entities: (*i*) `Process`, which represents the composite services deployed inside MOSES, providing a high-level description of the business processes and of their execution graphs; (*ii*) `User`, which represents the registered users; (*iii*) `AbstractService`, which represents the abstract part of the WSDL files and is complemented by the *AbstractOperation* weak entity; (*iv*) `ConcreteService`, which represents the actual services invoked by each business process, including its SLA parameters, and is complemented by the *ConcreteOperation* weak entity; (*v*) `SLA`, which represents the QoS classes offered for the corresponding composite service, including the QoS values agreed for each service class in the SLA and the actual perceived QoS values monitored during the MOSES execution; (*vi*) `Group`, which represents the coordination patterns (the currently supported ones are *single*, *alt*, and *par_or* [12]).

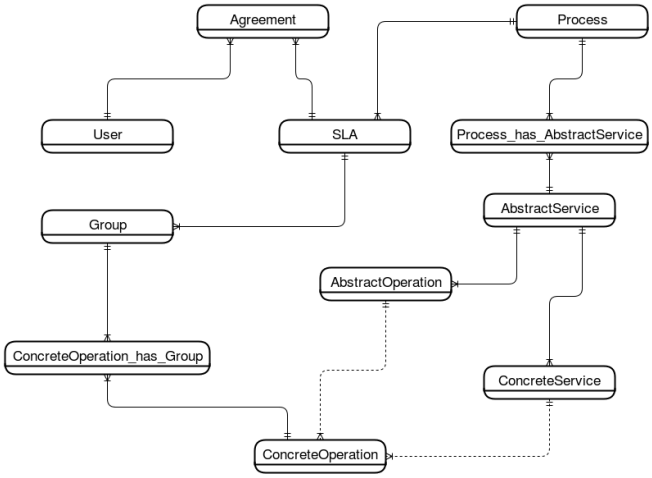


Fig. 4: Database ER diagram.

MOSES presently uses the relational database MySQL, which provides transactional features through the InnoDB storage engine and supports clustering and replication. We carefully optimized the MDAL methods to let the Adaptation Manager efficiently read the adaptation policy stored in the database, since this could happen even thousands of times per second. Since the solution of the optimization problem is scattered among multiple database tables, in order to avoid expensive and repeated table joins, MDAL creates a new `solution` table from temporary in-memory tables that are then dropped whenever a new optimal solution is computed. Furthermore, we improved the performance of executing queries to the database by exploiting the connection pool mechanism provided by Glassfish, which allows to cache database connections so that they can be reused when future requests to the database are required.

4.2 MOSES Extensions

MOSES has a modular and easily extendible architecture. It has been realized in such a way that the prospective developers could both easily change the behaviour of already developed components and add new currently unplanned functionalities. For instance, in order to develop a new service selection policy that uses the QoS parameters already supported by MOSES, the developer has just to implement with his/her own code the Optimization Engine Java interface and change the proper MOSES configuration file. The new class will be dynamically loaded at runtime. Such an extension does not require any modification to the other MOSES modules.

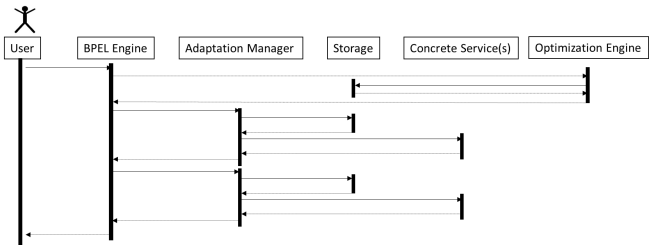
The addition of new functionalities may require instead to change the data model. For instance, if the prospective developer is interested in considering a service selection based on the actual network latency between MOSES and the concrete services, s/he must add that QoS attribute to the data model. Such a change must be then reflected into MDAL and a new MOSES module must be developed to make MOSES able to measure the new metric. The newly developed module can then be easily plugged into the architecture presented in Fig. 3, where it will be able to exploit other services, first of all the Optimization Engine. The latter must be also properly extended to handle the new QoS attribute. We observe that the modifications to MDAL needed for the new module and for the extended Optimization Engine do not impact the normal operativeness of all the other modules, which can still use the original version of the data access library.

Details on how to extend MOSES can be found in the documentation available at `http:/www.ce.uniroma2.it/moses/`.

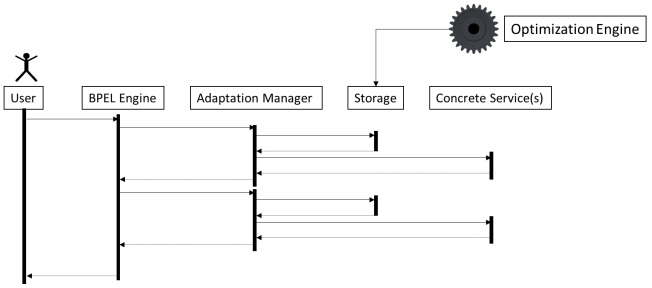
4.3 MOSES Overheads

As observed in Chap. XXX of this book (*Perpetual Assurances in Self-Adaptive Systems*), to assure compliance with the requirements for self-adaptive systems at runtime, the employed techniques must not interfere with the ability of the self-adaptive system to deliver its intended functionality effectively and efficiently. To achieve this goal, in the MOSES design and implementation we took care of the different types of overheads introduced by the runtime adaptation management. They can be classified according to the MOSES macro-components: (*i*) overhead due to the Plan macro-component; (*ii*) overhead of the Execution macro-component due to the runtime binding of the task endpoints to concrete implementations; (*iii*) overhead due to the Monitor and Analyze macro-components.

Plan Overhead The overhead introduced by the Optimization Engine depends on the time taken to calculate a new adaptation policy (and hence the class of optimization problem formulation) as well as on the synchronous or asynchronous invocation of the Optimization Engine with respect to the service execution flow, as illustrated in Fig. 5.



(a) Per-request adaptation policy.



(b) Per-flow adaptation policy.

Fig. 5: Interaction among the MOSES modules in the service execution flow.

As regards the optimization problem formulation, the overhead depends on the computational complexity of the optimization problem that drives the adaptation policy. We demonstrated in [12] that the adoption of a linear programming model as optimization problem helps considerably in terms of scalability with respect to other approaches in literature that rely instead on more computationally expensive formulations: specifically, our approach results from one to two orders of magnitude faster. For example, for problem instances of reasonably large size (1000 service tasks in the abstract composite service and 50 concrete services implementing each task), the LP problem formulation adopted by MOSES requires 8.64 s. to be solved, against 451.30 s. of the per-flow approach in [6] and 19.88 s. of the per-request approach in [4] (see Table 5 in [12] for the complete experiment). Nevertheless, the exploitation of redundancy patterns can result in excessive computational costs in case of large scale problem instances where the number of candidate services grow dramatically. Such costs can be restrained by either limiting the use of these patterns to a subset of the abstract tasks, or by limiting the maximal number of candidate services that can be used to implement a redundancy pattern. To this end, the redundancy degree is a tunable parameter within the per-flow policy of the Optimization Engine.

As regards the synchrony of the optimization problem solution and the service execution flow, we observe that the per-request adaptation policy requires to solve the optimization problem synchronously to each service request (see Fig. 5a), because no QoS class is defined a-priori and the required QoS attributes are set by the user for each request addressed to the broker. After having solved

the optimization problem, the corresponding adaptation policy is stored in the database and then used for the subsequent invocations pertaining to the same user request.

On the contrary, in the per-flow adaptation policy, the optimization problem is solved asynchronously with respect to the service execution flow (see Fig. 5b), while incoming service requests are served by the Adaptation Manager according to the previously calculated policy, which is stored in the database. Therefore, the computational cost of the optimization problem does not directly impact on the MOSES ability to respond to the user requests (because in the meanwhile the old, sub-optimal policy can be used), but only affects its responsiveness in updating the adaptation policy.

Execute Overhead The overhead in the Execute phase is related to the runtime binding of abstract tasks with concrete services carried out by the Adaptation Manager and affects each request to the composite service as many times as the number of `invoke` activities executed in the BPEL process, as shown in Fig. 5. 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 actual operation(s) (and possibly the coordination pattern) to implement the abstract task.

We measured the overhead added by the MOSES runtime binding to the execution of the GlassFish ESB engine, finding that the MOSES response time is on average 74% higher than that provided by the plain BPEL engine [12].

Monitor and Analyze Overhead We observe that the Analyze phase does not affect the overall service time perceived by a user, since the related operations are executed asynchronously with respect to the composite service.

The most time consuming and frequent monitoring activity is that performed with respect to the QoS parameters (specifically, the response time) offered by the concrete services. In this case, the MOSES monitoring overhead is about 1 ms for each `invoke` activity in the composite service, as it only involves inserting the operation response time in a database table: for each `invoke` activity, MOSES gets the timestamp before and after the invocation itself, calculates the observed response time and then puts it into the storage layer.

4.4 MOSES Evaluation Tool

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 implemented a workload generator in C language using the Pthreads library. The workload generator was designed to test the per-flow adaptation policy but can be as well as used for the per-request adaptation policies, as we did in [13]. It is based on an open system model, where users requesting a given service class k offered by MOSES arrive at mean *user inter-arrival rate* Λ_k. Each class k user u is characterized by its SLA parameters and by the *contract duration* t_u^k. Each incoming user is subject to an admission control, carried out by the SLA

Manager as discussed in Sec. 4.1. If the SLA Manager admits the user, it starts generating requests to the composite service according to the rate λ_u^k until its contract ends. Otherwise, the user terminates.

In the workload model of our generator 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. Such classic distributions can be easily changed in case new studies will evidence some peculiar characteristics of SOA traffic. The workload generator uses multiple independent random number streams for each stochastic component of the workload model. The generator reports as metrics for the composite service managed by MOSES the QoS attributes (i.e., response time, reliability, and cost) that are currently considered by MOSES.

To experimentally evaluate the performance of the self-adaption policies, we set up a composite service that mimics a travel planner [12]. The concrete services that we provide in the MOSES package are simple stubs, without internal logic; however, their non-functional behavior can be easily set to conform to the guaranteed levels expressed in their SLA.

5 Related Work

Several solutions have been proposed for the self-adaptation of SOA systems, mainly focusing on the runtime service selection (e.g., [3, 5, 6, 4, 10, 14, 16, 17, 22, 23, 32, 37, 38]. Service selection has been and still is a challenging problem because the number of candidate services offering the same functionalities but differing in QoS is increasing with the prevalence of service-based systems and Cloud computing. However, only few frameworks and platforms have been designed, implemented and evaluated in the scope of self-adaptation of SOA systems. To the best of our knowledge, the platforms that share some common features with MOSES include MUSIC [35], SASSY [28, 26], VieDAME [31], DISCoRSO [5], and VRESCo [29]. In this section we review their characteristics and features according to the benchmarking criteria proposed in Chap. XXX of this book (*Perpetual Assurances in Self-Adaptive Systems*), which are summarized in Tab. 1.

MOSES handles requirement variability by solving an optimization problem, aimed at maximizing a utility function subject to constraints given by the current operating environment. It implements two classes of optimization problems, named *per-flow* and *per-request* as described in Sec. 4.1. In the per-flow scenario the application architect is expected to define the QoS classes that MOSES will advertise and offer to perspective users. Conversely, in the per-request scenario, no QoS class is defined a-priori and the required QoS attributes must be set by the user for each request addressed to the broker. As a consequence, in the former scenario MOSES can compute a-priori an optimal adaptation strategy for each service class defined by the application architect and subsequent re-optimizations are only due to changes in the environment or to user arrival/departure. In case

Table 1: Summary of assurance characteristics of existing frameworks

Benchmark aspect	Criteria	MOSES	MUSIC	SASSY	VieDAME	DISCoRSO
	Variability	✓	✓	✓	✓	✓
	Inaccuracy & incompleteness	✓	✓	✓	✓	✓
Capabilities of approaches to provide assurances	Conflicting criteria	✓	✓	✓	✓	✓
	User Interaction		✓			
	Handling alternatives	✓	✓	✓	✓	✓
Basis of assurance benchmarking	Historical data	✓	✓	✓	✓	✓
	Projections in the future					✓
	Human evidence		✓			
Stringency of assurances	Assurance rational	✓				✓
Performance of approaches	Timeliness	✓				
	Computational overhead	✓		✓	✓	
	Complexity	✓		✓		✓

the user tries to establish a service contract when MOSES has not sufficient available resources, the optimization problem will result unfeasible, thus leading to a rejection of the new contract. On the other hand, in the per-request scenario MOSES has to compute a new adaptation policy for each request it receives, being each request characterized by its own QoS requirements. Should the user require too stringent QoS attributes, the optimization problem will be unfeasible, thus leading to the request rejection. Conflicting optimization goals are managed by MOSES using the SAW technique in the utility function. Since MOSES is not designed for long lasting processes, user interaction is not expected at runtime, but adaptation policy preemption is possible in the per-flow adaptation strategy. MOSES continuously monitors service execution time and reliability. The collected data is then analyzed to discover possible SLA violations and therefore to trigger a new optimization request. MOSES provides strict QoS assurances in case of the per-request adaptation strategy, while it is capable of assuring average QoS attributes on a flow of request with the per-flow adaptation strategy. From the performance perspective, MOSES is able to provide a runtime dynamic binding with a very small overhead (more details are provided in Sec. 4.3), while the optimization process, in the per-flow arrangement, is designed as a LP problem, which can be solved in polynomial time. The per-request optimization problems are formulated as MILP problem, thus having a NP-hard complexity.

MUSIC handles requirements variability by adapting the applications it manages according to the current context (e.g., the availability of certain resources such as GPS signal and WiFi networks). Inaccuracy and incompleteness at design time of what will be the runtime environment are managed through a constant monitoring activity: MUSIC is able to detect the health state of known services and, should any of them not being compliant with the negotiated QoS levels, it can be replaced by some other known service. Furthermore, MUSIC can discover new services and integrate them seamlessly in the service composition logic. Differently from the other considered platforms, MUSIC is able to concurrently manage several applications on a single device (e.g., a smartphone). Therefore,

given the inherent constraints of the device it runs on, it must be able to maximize a utility function that addresses all the QoS parameters of every running application, trying to provide a weighted performance compromise among them. For this behavior to be effective, the user needs to interact with the framework in order to choose, for instance, the service class for a given application. Both human evidences and monitored historical performance data are used to determine whether to compute a new adaptation strategy. The framework, however, is not thought for mission-critical applications and there is no assurance it will run with an optimal configuration, because the authors did not formulate any optimization problem. Rather, the best possible configuration is chosen from a set of possible configurations after evaluating a utility function; however, how to populate the set of possible configurations is not discussed.

SASSY is a model-driven framework that provides runtime adaptation of service compositions in response to system variations as those caused by services violating the negotiated SLA. It adopts a MAPE-K loop to monitor the exploited services, analyze monitored data, plan a new implementation of the application, and finally execute the application. SASSY allows the application architect to specify at design time the QoS attributes desired for the application, which are then used to build a runtime model that is then exploited to achieve the QoS goals. Possibly conflicting criteria are managed by weighting a utility function and optionally adding constraints like cost. User interaction is not expected at runtime: the entire framework is designed to let the application architects specify their QoS constraints at design time. However, given such QoS constraints, new alternatives are automatically evaluated in response to changing operating conditions (e.g., new services are discovered or existing services violate their SLA). The reactive behavior of SASSY is backed by an analysis of data regarding the performance of the exploited services. Since the formulated optimal service selection problem is NP-hard, a heuristic based on hill-climbing was proposed to solve it efficiently. As a consequence, there is no guarantee that the actual instantiated architecture will strictly satisfy the required QoS.

VieDAME is able to handle variability: it decouples the required functionalities from the actual concrete services and provides a runtime dynamic binding. The selected concrete services are then continuously monitored in order to verify whether they satisfy the agreed SLA. Should VieDAME find some violation, the bound implementations can be replaced, according to the specific service selector used to implement the dynamic binding. Furthermore, VieDAME addresses the problem of carefully designing domain-specific QoS requirements by providing a simple language (named VieDASSL) that can be exploited by domain experts to define QoS requirements. VieDAME service selection is based on a score computed on the basis of some service selection rules named *Factor rules*: the simplest is the *Simple Factor Rule*, which is able to manage at most one QoS attribute at a time, thus not providing a tool to manage conflicting criteria. The latter are addressed with the *Sum Selection Rule*, which provides to the domain expert a tool to describe a weighted sum of QoS attributes for the implementation of a given functionality. Alternative services are considered for

two purposes: load balancing and SLA violations. VieDAME is built on top of a MAPE-K loop for autonomic systems which continuously monitors the perceived QoS attributes of every concrete service involved and, based on historical data, it determines whether to replace the services. VieDAME does not support strict fulfillment of QoS attributes: it exploits VieDASSL to define rules that are then used to drive the computation of scores for the available services. Such scores are finally used to build a classification of the services and the best one is chosen to implement the required functionality. To avoid overloading the best service, a selection post-processor is also implemented: instead of always using the same best concrete service, a subset of the available services with a score greater than a threshold is considered. The CPU overhead introduced by the VieDAME runtime binding and monitoring is in the range of 10%-20% with respect to the execution of the same BPEL process without self-adaptation features.

DISCoRSO lets the application architects define both local and global QoS constraints. During the application execution, QoS attributes are continuously monitored in order to detect possible SLA violations and re-optimize the whole execution process. To this end, monitored data is stored into the database and then used by the analyzer module which is the responsible of SLA violation detections. DISCoRSO uses a mix of historical data analysis and projections in the future to minimize the probability of not satisfying the SLA of the whole application, also by eventually triggering a pre-emptive live re-optimization. This behavior is particularly useful to provide QoS assurances on long-term executions. The optimization problem is formulated as a MILP problem (thus having NP-hard complexity) and supports the management of possibly conflicting criteria. Its optimal solution, which must be computed for every invocation to the composite application, supports a strict fulfillment of the required QoS attributes.

VRESCo neither proposes nor implements optimal or sub-optimal solutions to provide an adaptation strategy: it rather provides a framework for dynamic binding and a query language that can be used to retrieve execution plans that must be afterwards processed by an optimization algorithm. Therefore, VRESCo cannot provide by itself assurances on QoS attributes, rather it provides an execution platform which can be enriched by adding optimization features. Specifically, referring to MAPE-K loop, it only implements the Execute phase. However, VRESCo is the only considered platform which explicitly supports Service Mediation: it accepts from the user a high-level representation of the data that will be used as input for the concrete services. These data are then lowered (i.e., transformed from high-level representation into low-level format) in order to be compatible with the given concrete service. The response is instead lifted (i.e., transformed from low-level format to high-level representation) in order to be compatible with user expectations. From a performance perspective, the VRESCo runtime binding adds about 400 ms to the service execution time, without considering the service mediation. This additional time is due to the addition of a proxy which, for each service invocation, queries the database in order to know which service to invoke.

6 Conclusions

We have presented MOSES, a software platform supporting QoS-driven adaptation of service-oriented systems. MOSES is architected as a broker that controls the self-adaptation of a composite service by implementing the functionalities of a MAPE-K control loop. The modular MOSES architecture facilitates the integration in the overall platform of different adaptation policies and mechanisms, thus making it a suitable testbed for their experimentation, thanks also to the availability of the complete platform source code.

As described in the previous sections, MOSES provides a complete implementation of the core functionalities of the MAPE-K loop, tailored for the SOA environment. However, we remark that some useful functionalities, which would make MOSES a comprehensive solution for the management of SOA systems, are currently not included in the MOSES distribution. They include, in particular: (*i*) the negotiation of SLAs with the prospective users of the managed composite service, and with the providers of services exploited by MOSES; (*ii*) the discovery of services to be included in the pool of possible candidates for the composition of the managed service; (*iii*) the automatic derivation of the optimization problem that constitutes the core of the MOSES strategy, from a model of the composite service, QoS requirements and constraints.

The QoS-driven self-adaptation policies provided by MOSES suffer from two limitations: first, they do not consider the network latency and bandwidth between the broker and the concrete services; second, they use point estimates of the QoS attributes of the concrete services which may not correctly reflect the actual statistics, e.g., under high variance the point estimates may take too long to converge to the actual value. We plan to address the first issue by including network parameters within the MOSES model and devising a new formulation of the optimization problem. We will tackle the second issue by adopting robust optimization techniques [9] that are capable to provide solutions which satisfy the composite service QoS requirements despite the QoS attributes uncertainty and/or variability.

As a final remark, we note that MOSES itself is a complex system that could benefit from the addition of self-adaptation features to automatically scale in/out in response to variations in the load of user requests it has to manage, or to adjust its reliability. This extension of the MOSES platform is subject of ongoing work.

Acknowledgments V. Cardellini and F. Lo Presti acknowledge the support of ICT COST Action IC1304 ACROSS.

References

1. Abundo, M., Cardellini, V., Lo Presti, F.: Admission control policies for a multi-class qos-aware service oriented architecture. ACM SIGMETRICS Perform. Eval. Rev. 39(4), 89–98 (2012)

2. Alférez, G., Pelechano, V., Mazo, R., Salinesi, C., Diaz, D.: Dynamic adaptation of service compositions with variability models. J. Syst. Softw. 91, 24–47 (2014)
3. Alrifai, M., Risse, T.: Combining global optimization with local selection for efficient qos-aware service composition. In: Proc. WWW '09. pp. 881–890 (2009)
4. Ardagna, D., Pernici, B.: Adaptive service composition in flexible processes. IEEE Trans. Softw. Eng. 33(6), 369–384 (2007)
5. Ardagna, D., Baresi, L., Comai, S., Comuzzi, M., Pernici, B.: A service-based framework for flexible business processes. IEEE Software 28(2) (2011)
6. Ardagna, D., Mirandola, R.: Per-flow optimal service selection for web services based processes. J. Syst. Softw. 83(8), 1512–1523 (2010)
7. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: Issue and challenges. IEEE Computer 39(10), 36–43 (2006)
8. Bellucci, A., Cardellini, V., Di Valerio, V., Iannucci, S.: A scalable and highly available brokering service for SLA-based composite services. In: Proc. ICSOC '10. Lect. Notes Comput. Sc., vol. 6470, pp. 527–541. Springer (2010)
9. Ben-Tal, A., El Ghaoui, L., Nemirovski, A.: Robust Optimization. Princeton University Press (2009)
10. Berbner, R., Spahn, M., Repp, N., Heckmann, O., Steinmetz, R.: Heuristics for QoS-aware web service composition. In: Proc. IEEE ICWS '06. pp. 72–82 (2006)
11. Canfora, G., Di Penta, M., Esposito, R., Villani, M.: A framework for QoS-aware binding and re-binding of composite web services. J. Syst. Softw. 81(10), 1754–1769 (2008)
12. Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Lo Presti, F., Mirandola, F.: MOSES: A framework for QoS driven runtime adaptation of service-oriented systems. IEEE Trans. Softw. Eng. 38(5), 1138–1159 (2012)
13. Cardellini, V., Di Valerio, V., Grassi, V., Iannucci, S., Lo Presti, F.: QoS driven per-request load-aware service selection in service oriented architectures. Int'l J. of Software and Informatics 7(2), 195–220 (2013)
14. Chen, Y., Huang, J., Lin, C.: Partial selection: An efficient approach for QoS-aware web service composition. In: Proc. IEEE ICWS '14 (2014)
15. Erradi, A., Maheshwari, P., Tosic, V.: WS-policy based monitoring of composite web services. In: Proc. ECOWS '07. pp. 99–108. IEEE (2007)
16. He, Q., Yan, J., Jin, H., Yang, Y.: Quality-aware service selection for service-based systems based on iterative multi-attribute combinatorial auction. IEEE Trans. Softw. Eng. 40(2), 192–215 (2014)
17. Huang, J., Liu, G., Duan, Q., Yan, Y.: Qos-aware service composition for converged network-cloud service provisioning. In: Proc. IEEE SCC '14. pp. 67–74 (2014)
18. Hwang, C., Yoon, K.: Multiple Criteria Decision Making, Lect. Notes Econ. Math., vol. 186. Springer (1981)
19. IBM Corp.: CPLEX Optimizer (2016), `http://www.ibm.com/software/commerce/optimization/cplex-optimizer/`
20. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
21. Klein, A., Ishikawa, F., Honiden, S.: Efficient QoS-aware service composition with a probabilistic service selection policy. In: Proc. ICSOC '10. Lect. Notes Comput. Sc., vol. 6470, pp. 182–196. Springer (2010)
22. Klein, A., Ishikawa, F., Honiden, S.: SanGA: A self-adaptive network-aware approach to service composition. IEEE Trans. Serv. Comput. 7(3), 452–464 (2014)
23. Leitner, P., Hummer, W., Dustdar, S.: Cost-based optimization of service compositions. IEEE Trans. Serv. Comput. 6(2), 239–251 (2013)

24. MathWorks Inc.: MATLAB® (2016), http://www.mathworks.com/
25. Menascé, D., Casalicchio, E., Dubey, V.: On optimal service selection in service oriented architectures. Perform. Eval. 67(8), 659–675 (2010)
26. Menascé, D., Gomaa, H., Malek, S., Sousa, J.P.: Sassy: A framework for self-architecting service-oriented systems. IEEE Software 28(6), 78–85 (2011)
27. Menascé, D.A.: Qos issues in web services. IEEE Internet Comp. 6(6), 72–75 (2002)
28. Menascé, D.A., Ewing, J.M., Gomaa, H., Malex, S., Sousa, J.P.: A framework for utility-based service oriented design in SASSY. In: Proc. WOSP/SIPEW '10. pp. 27–36. ACM (2010)
29. Michlmayr, A., Rosenberg, F., Leitner, P., Dustdar, S.: End-to-end support for QoS-aware service selection, binding, and mediation in VRESCo. IEEE Trans. Serv. Comput. 3(3), 193–205 (2010)
30. Montgomery, D.: Introduction to Statistical Quality Control. Wiley (2008)
31. Moser, O., Rosenberg, F., Dustdar, S.: Domain-specific service selection for composite services. IEEE Trans. Softw. Eng. 38(4), 828–843 (2012)
32. Ngoko, Y., Goldman, A., Milojicic, D.: Service selection in web service compositions optimizing energy consumption and service response time. Journal of Internet Services and Applications 4(1) (2013)
33. OASIS: Web Services Business Process Execution Language Version 2.0 (2007)
34. OpenESB - The Open Enterprise Service Bus (2015), http://www.open-esb.net/
35. Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., Scholz, U.: MUSIC: Middleware support for self-adaptation in ubiquitous and service-oriented environments. In: Software Engineering for Self-Adaptive Systems, pp. 164–182. Springer (2009)
36. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 1–42 (2009)
37. Schuller, D., Siebenhaar, M., Hans, R., Wenge, O., Steinmetz, R., Schulte, S.: Towards heuristic optimization of complex service-based workflows for stochastic QoS attributes. In: Proc. IEEE ICWS '14. pp. 361–368 (2014)
38. Zivkovic, M., Bosman, J., van den Berg, H., van der Mei, R., Meeuwissen, H., Nunez-Queija, R.: Run-time revenue maximization for composite web services with response time commitments. In: Proc. IEEE AINA '12. pp. 589–596 (2012)