PAWS: a framework for processes with adaptive web services

Danilo Ardagna, Marco Comuzzi, Enrico Mussi, Pierluigi Plebani, Barbara Pernici
Politecnico di Milano
contact author: comuzzi@elet.polimi.it

Abstract

The paper introduces PAWS (Processes with Adaptive Web Services), a framework for flexible and adaptive execution of managed Web service-based business processes. In the framework several modules for service adaptation are integrated in a coherent way. An original characteristic of this framework is to couple design-time and run-time mechanisms for process specification and execution in a global framework. At design-time, flexibility is achieved through a number of mechanisms, i.e., identifying a set candidate services for each process tasks, negotiating quality of service, specifying quality constraints, and identifying mapping rules for invoking services with different interfaces. In turn, the run-time exploits the design-time mechanisms to support adaptation during process execution, in terms of selecting the best set of services to execute the process, reacting to a service failure, or preserving the execution when a context change occurs. The paper also discusses how PAWS has been applied in several case studies.
Keywords: Service-based processes, QoS, adaptivity.

1 Introduction

The vision of self-managed applications is being proposed in autonomic computing [**?**] to support self-managing computing systems, where system components are self-configuring, self-optimizing, self-healing, and self-protecting. Autonomic computing applied to business processes based on Web services (hereafter simply called services) is emerging as a new research area. The result is the development of *flexible* and *adaptive* business processes based on the service oriented approach. We define flexibility as the capability of changing process behavior dynamically according to variable execution contexts; adaptivity is the capability of executing a service even when the ex-

ecution conditions are not exactly the ones assumed during the initial design of that service, and the executed service might offer limited functionalities.

This paper presents PAWS (Processes with Adaptive Web Services), a framework for flexible and adaptive execution of managed service-based processes which supports in a coherent way both process design and its execution. This framework is the result of the integration of a number of research results developed at Politecnico di Milano considering different aspects of adaptation, coupling design-time and run-time mechanisms in a global environment. The goal of the PAWS framework is twofold. On one hand, the framework supports optimized service selection and the definition of the most appropriate Quality of Service (QoS) levels for the selected services (self-optimizations). On the other hand, PAWS aims at guaranteeing the service provisioning also in case of failures, managing services through recovery actions whose goal is to continue service execution when a fault is detected (self-healing) and to self-adapt to context changes. Therefore, the PAWS approach proposes methods and a toolset to support at *design-time* the specification of all information required to support at *run-time* the automatic process adaptation according to dynamically changing users' preferences and context [**?**, **?**].

The modules included in the PAWS framework support a number of flexibility and adaptation mechanisms. While several adaptation mechanisms have been proposed in the literature, there is a lack of an integrated framework which systematically couples adaptation design and run time execution. In particular, we focus on selecting and adapting candidate services for a composed process, providing in addition annotations to exploit flexibility at run time, where optimization, mediation, and self-healing functionalities are provided. The framework is general and can be extended by introducing domain-dependent service annotations, with QoS and context definitions.

Papazoglou et al. [**?**] advocate the need of extending traditional Service Oriented Architecture also considering service composition and service management. Adaptive mechanisms for workflows and service-based processes have been proposed in the literature. In Mosaic [**?**] a framework is proposed for modeling, analyzing, and managing service models; in this work the focus is on de-

sign phases rather than flexibility at run-time. Meteor-S [**?**] and other semantic-based approaches [**?**] explicitly define the process *goal* on which they perform both service discovery and composition. In Meteor-S a framework is proposed to select semantically annotated services focusing on the flexible composition of the process and also on QoS properties, but run time adaptivity to react to changes and failures is not considered. In WSMO [**?**], a goal-based framework is proposed to select, integrate, and execute semantic Web services. No separation between design and run time phases is proposed, nor specific support to design adaptivity. However, while goal-based approaches open up the possibility of deriving service compositions at run time, their applicability in open service-based applications is limited by the amount of knowledge available in the service definitions.

Other literature [**?**, **?**] proposals tackle single aspects of adaptations. In business process optimization approaches, the process specification is provided and the best set of services is selected at run time by solving an optimization problem [**?**]. In a similar way, in grid systems applications are modelled as high level scientific workflows where resources are selected at run time in order to minimize, for example, workflow execution time or cost. Anyway, even if run-time re-optimization is performed and provides a basic adaptation mechanism, user context changes and self-healing are not addressed. In PAWS we support adaptation preferences coupled with adaptation execution in a coherent framework focussing both on functional and QoS properties. With respect to self-healing mechanisms, the PAWS framework allows adaptation and flexibility at run time based on optimization and negotiation mechanisms and predefined repair actions.

The PAWS framework and its adaptation mechanisms are illustrated in Section 2, whereas the different modules are illustrated in detail in Section 3. The framework has been applied to a number of case studies, including mobile services in ad hoc networks, production processes, and emergency management. Some considerations on the applicability of the framework modules in different scenarios are illustrated in Section 4.

2 PAWS framework

Figure 1 shows the modules which compose the PAWS framework, classifying them in *design-time modules* and *run-time modules*. The aim of design-time modules is the specification of a service-based process defined with an annotated version of BPEL (step 1). Annotations identify, for each task, the set of candidate services obtained after a *discovery* phase operated on a service registry (step 2). In addition, annotations also include both *local* and *global* constraints. Local constraints express QoS or domain-dependent requirements for each task, while global constraints can be associated to the whole process, to define requirements on QoS or dependencies among component tasks. Such a BPEL process definition is deployed in a BPEL engine, enhanced with PAWS run-time modules (step 3). As long as the user invokes the deployed BPEL process (step 4), the run-time modules support the BPEL process execution by dynamically *selecting*, for each task, the most suitable service among the set of candidates and adapting their interfaces if needed (step 5). Moreover, in case of failure, run-time modules can also recover the process execution according to a set of recovery rules.

Both service discovery and service selection are driven not only by functional aspects (what the service should do), but also by non-functional aspects (how the service should work). About the latter, all PAWS modules rely on a shared QoS model [**?**] to express global and local QoS constraints, to discover the set of candidate services, and for selecting the most suitable services.

The QoS model aims at providing an extensible way to express the quality of both tasks and the whole process. A QoS dimension represents a specific quality aspect, e.g., response-time, cost, availability. A quality dimension is specified by a name, a metric, a range of admissible values (either interval or categorical) and by an utility function stating how the QoS varies with respect to the values assumed by the dimension. Domain dependent QoS properties can also be specified. Service providers publish their services in the registry also describing their QoS.

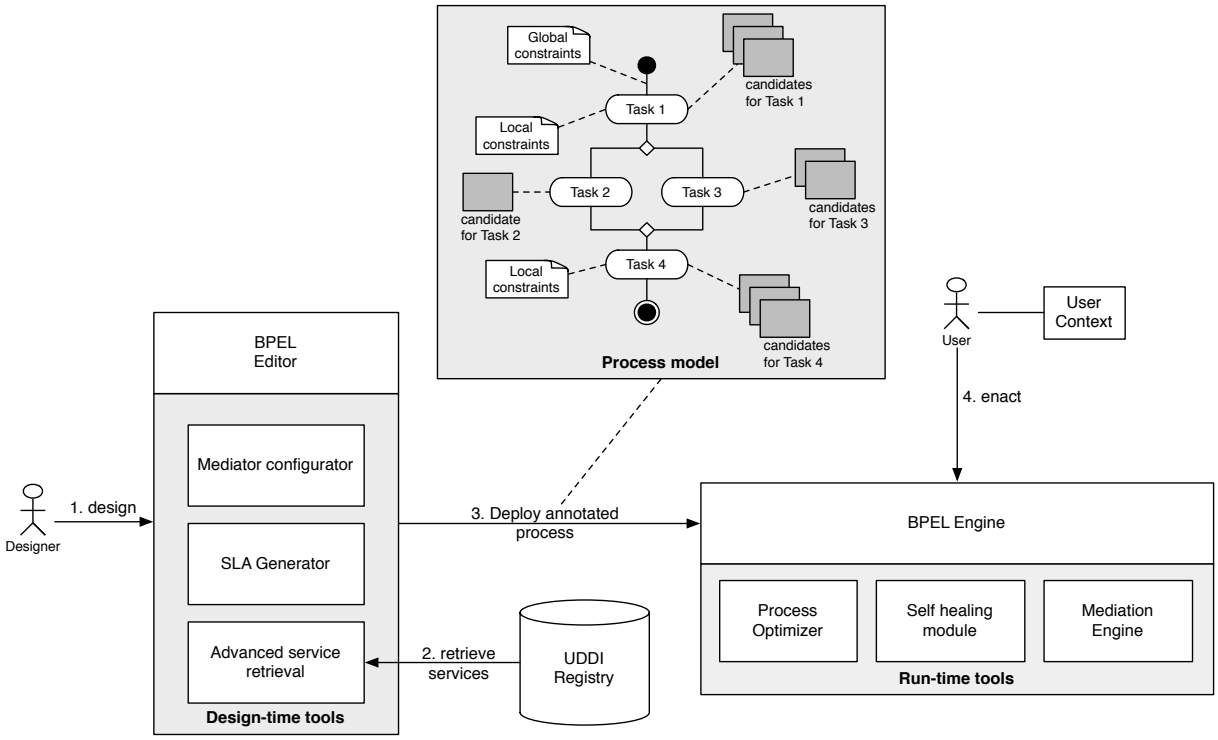

Figure 1: PAWS architecture

2.1 Design-time modules

Design-time modules allow the designer to create the annotated BPEL process specification. Starting from a standard BPEL Editor, the designer can define both global and local constraints. A constraint defines a QoS or domain-dependent requirement for each task, i.e. a constraint defines how the corresponding service must work. Throught the annotation of the BPEL process, the designer couples the functional requirements expressed by the BPEL specification with the constraints in PAWS which usually refer to non-functional aspects. PAWS design-time modules allow the designer to define the process according to a top-down approach, starting from the desiderata expressed by the annotated BPEL. For instance, the designer can require that a given task must be performed in a given time, or the overall cost cannot be more than a given budget.

In traditional design approaches [**?**], the designer starts identifying the potential partners and then he/she defines the BPEL process starting from the WSDLs previously selected. On the con-

trary, with PAWS, the designer can initially focus on the process definition and then he/she can select the services able to perform the required invocations and satisfying the constraints as expressed by annotations. In such a scenario, given a task, the designer relies on the *Advanced Service Retrieval Component* to retrieve all the published services able to perform a given task comparing the required service interface (defined in WSDL) with the published ones. The retrieval component also verifies that the QoS supported by the published service satisfies the local constraints. The set of services passing both analyses will constitute the candidates for the considered task.

From a functional perspective, the retrieval process returns services similar to the desired one and, only in some rare cases, it returns services with a WSDL interface exactly matching the requested one. Thus, mediators able to translate between the two service interface signatures are required. To this aim, the PAWS framework includes a *Mediator Configurator* to support the designer to set up the related run-time module, i.e., the *Mediator Engine*, able to perform the required transformations of messages during the process execution, providing additional information about parameter and service mappings, in case it is not possible to derive them automatically.

Concerning non-functional constraints, a service is included in the candidates set if it is able to support the required QoS. Before invoking the service, the designer assisted by the *SLA (Service Level Agreement)* Generator defines the actual QoS that the service commits to support during the process execution and that will be monitored at run time.

2.2 Run-time modules

In PAWS the process definition does not include invocations of real services, but of desired services, as characterized by the annotated process. Run-time modules select, for each task, one service among the candidates. Moreover, since the process is executed by a traditional *BPEL Engine*, and the services to be invoked have been selected at run-time, run-time modules also mediate between the BPEL invocation – as specified in the process definition – and the invocation of the

selected service.

Service selection is performed by the *Process Optimizer* which aims at (i) satisfying both the local and the global QoS constraints expressed by the annotations, (ii) maximizing the overall quality for the user. User preferences are collected in the *User context*, which transparently exchanges information on context between the user and the framework. Context in PAWS is represented with <name,value> pairs, thus allowing for defining generic and domain-dependent characteristics, such as user-defined QoS priorities, negotiation preferences, user location information, and the like. The details of context management are out of the scope of the present paper and are discussed in [**?**].

Service mediation is achieved by the *Mediation Engine*, suitably set-up by the *Mediator configurator* at design-time. This module is in charge of redirecting the invocation performed by the deployed process to the proper selected services.

As long as the process execution works fine, no additional efforts are required. On the contrary, in case of failure of the participating services or of the process itself, adaptation is handled by the *Self-healing* module. This module implements a set of semi-automatically managed recovery actions, that enhance the process adaptivity when a fault during the process execution is detected. In some cases, the failure implies a service substitution for a given task. In such a case, the *Process Optimizer* is involved again to select services from the set of candidates for the remaining tasks to be executed in order to guarantee global constraints.

3 Components for flexibility and adaptivity

In the present section, the functionalities of each PAWS module are illustrated. Design time modules provide: (i) advanced service retrieval (Sect. 3.1) and negotiation support (Sect. 3.2). At run time, business process optimization selects the set of services considering the user context (Sect. 3.3). Mediation support modules are provided to configure mediation functionalities at design time and to perform mediation activities (Sect. 3.4) and self-healing functionalities (sect. 3.5) at run

time.

3.1 Advanced service retrieval

Service retrieval is performed by URBE (Uddi Registry By Example): a UDDI extension supporting content-based queries. Unlike what happens with current registries [**?**], with URBE the designer can find a service not only browsing pre-defined taxonomies, or by keywords, but also submitting a WSDL specifying the requested service in terms of supported operations and input and output parameters. As a result, URBE returns the set of published services more similar to the requested WSDL interface [**?**].

URBE is composed by a functional and a QoS matchmaker. The functional matchmaker is driven by a similarity evaluation algorithm able to state the similarity among services based on their descriptions. This algorithm starts with the assumption that two services are equal when both they require the same information as input and they produce same data as output for the same operations. The similarity algorithm works with the service signatures expressed by WSDL, taking into account both naming and structural aspects. On the one hand, naming aspects refer to the names adopted for identifying the service, the available operations, and the related exchanged parameters. On the other hand, the structural aspects refer to the number of operations available and the data type of the input/output parameters.In particular, name comparison relies on an ontology where terms are organized according to semantic relationships, such as synonym, antonym, homonym, hypernym [**?**]. A general purpose ontology, e.g., WordNet, is adopted, possibly extended with domain-specific terms and relationships. In addition, if available, the algorithm can also process SAWSDL (proposed by W3C, http://www.w3c.org) descriptions, where specific semantic annotations on WSDL elements are available as well.

The QoS matchmaker is invoked after the functional one and only if the designer requirements include QoS constraints as well. According to the model for expressing QoS in PAWS, this QoS matchmaker verifies whether a request and an offering match, i.e. if the offering is able to support

the request. At this stage, WS-Policy is adopted as the language for QoS offerings.

Experimental evaluations have validated the similarity algorithm in URBE. In some cases, precision and recall are about 80% [**?**].

3.2 Negotiation

The SLA generator allows the automated negotiation of SLAs on QoS aspects between the user and service providers for the candidate services identified by service discovery. The SLA generator only negotiates the QoS of candidate services for tasks with local budget constraints. Indeed, local budget constraints can be solved at design-time, i.e., the user and the provider can negotiate which is the maximum QoS that can be provided while satisfying the user's budget constraint. Performing negotiation automatically at design-time saves time and simplifies the run-time process optimization phase. Therefore, for each task for which a local budget constraint is specified, the SLA Generator module is in charge of negotiating the SLA for all the candidate services retrieved through URBE.

The objective of the negotiation is to obtain a SLA within the match of request and service offerings identified by the service discovery phase. Specifically, given a range for a QoS dimension, our QoS model supports the definition of a set of negotiation intervals, over which a pricing model can be specified by service providers. The pricing model is additive, since the final price associated to a service is evaluated as the sum of partial prices associated to each single QoS dimension interval. The budget specified in the designers' local constraints is used in the negotiation to improve the quality of the candidate service, starting from a basic SLA identified by the lowest QoS interval for each relevant QoS dimension. Different strategies can be adopted, such as splitting the budget proportionally to the users' priorities (*horizontal* strategy) or exploiting the budget to maximize the QoS of the dimension associated to the highest priority (*vertical* strategy).

The negotiation of the SLA for a single candidate service exploits two configuration policies expressed, respectively, by the service provider and by the designer, who acts on the behalf of the

user. The user policy contains the preferences over the QoS dimensions, expressed as a vector of weights, and the identifier of the SLA negotiation strategy. On the other hand, the service provider policy contains the parametrization of the pricing model for the provided service. After having parsed these policies, the SLA generator can automatically perform the SLA negotiation.

The negotiated QoS profiles can be updated at runtime by the Process Optimizer module, in case a feasible solution to the process optimization problem does not exist. The runtime negotiation occurs through the exploitation of an extra budget obtained from the global budget constraint. This runtime negotiation exploits a structure of preferences over QoS dimensions and service pricing models similar to the ones specified for the SLA negotiation at design-time.

We conducted an experimental evaluation of the two negotiation strategies by defining quasi-linear utility functions for users and providers with different combination of budget constraints and pricing models. The outcome of the horizontal negotiation strategy proved to be extremely close to the Pareto frontier of the negotiation problem. This result is noticeable, since the negotiation strategies are heuristics and negotiation is performed under the assumption of incomplete information among participants [**?**].

3.3 Business Process Optimization

Given the set of candidate services for each task in the process, the service selection is modeled as an optimization problem and takes into account end-user preferences, process global QoS constraints, and the run-time execution context. Furthermore, service selection and execution are interleaved: optimization is performed when the business process is instantiated and its execution is started. Optimization is also iterated during process execution performing *re-optimization* to take into account service performance variability, invocation failures, and user context changes.

In the literature, a number of solutions have been proposed to guarantee global constraints only for the critical path (i.e., the path which corresponds to the highest execution time) [], or to satisfy global constraints only statistically [**?**]. Furthermore, in such approaches, if the end-user

introduces severe QoS constraints for the composed service execution, i.e., limited resources which set the problem close to infeasibility conditions (e.g. limited budget or stringent execution time limit), no solutions could be identified and the composed service execution fails.

In our framework, we have implemented a new optimization approach based on mixed integer linear programming models, which overcomes the limits of the previous solutions optimizing the QoS perceived by the user under severe QoS constraints. Our modeling approach is based on loops peeling (which consider the probability distribution of the loops number of iterations) and negotiation, which is exploited if a feasible solution cannot be identified, to bargain QoS parameters with service providers, reducing process invocation failures.

The optimizer ranks the set of candidate services for each task (excluding services which lead to constraints violation), thus selecting a service for each task and ranking other candidate services which can be invoked as substitute services in case of a failure.

In [**?**], we have shown that our joint optimization and negotiation approach is effective in particular for large processes (up to 10,000 tasks), when QoS constraints are severe and it also reduces the re-optimization overhead. With respect to previous literature approaches, our algorithms improved the optimization problem objective function up to 40%.

3.4 Mediation Support

The goals of the *Mediation Engine* are: (i) supporting the invocation of services where a generic candidate service is dynamically bound without the need of compiling stubs at design time, (ii) managing service substitution. The substitution involves services which are described by possibly different signatures, but having the same choreography.

Whenever the BPEL Engine invokes a task, the Mediator uses the ranked list of candidate services previously generated by the Optimizer, taking the first service in the list. If the interface of the candidate service is different from the interface required by the task definition, the Mediator first retrieves the proper mapping document produced by the Mediator Configurator, and then invokes

the candidate service by sending transformed messages. The invocation of candidate services is managed by means of sessions that, in case a task is executed more than once, avoids the need to repeatedly access the ranked list and allow stateful service execution.

The Self-Healing module uses the Mediator to perform recovery actions, either to retry/redo the execution of a process task or to substitute a faulty candidate service. In case a candidate service has to be substituted, the Mediator first checks whether the context of the user has changed since the last optimization. In that case, it notifies the context change to the Optimizer that will generate a new ranked list of candidate services. Otherwise, the Mediator retrieves the next candidate service from the list provided by the Optimizer and then waits for the Self-Healing module to restart the process. In both cases, if a candidate service is no longer available, an exception is thrown.

For what concerns service substitution, the Mediator is able to accept messages formatted according to the interface of the substituted service and to translate them into messages formatted according to the interface of the substitute one by means of the Mediator Configurator.

3.5 Self-Healing

The self-healing behavior is conceived as a combination of monitoring and repair capabilities and its purpose is to detect if any failure arises during the execution of a process and, in case, to apply the appropriate recovery actions to let the process successfully terminate.

The recovery actions the Self-Healing module is able to perform are: (i) *retry* the execution of a process task, (ii) *redo* the execution of a process task by using different input parameters, (iii) *substitute* the currently used candidate service with another candidate, (iv) *compensate* an executed task with a compensation action defined within the service management interface.

The *retry* and the *redo actions* are usually used to recover from temporary faults, while the *substitute* action is used when an instance of an orchestrated service is considered as permanently faulty. *Compensation actions* are used to restore a previous state of the process. Since recovery actions cannot be performed over running process instances, once a fault has been detected, the

process is suspended and moved into a *repair mode*. After that all the necessary recovery actions have been performed, the instance is resumed and the process returns into the *running mode*.

The recovery actions are performed through the Mediation Engine. In particular, the *retry* and the *redo* are performed by explicitly asking the Mediation Engine to execute an already executed process task, while the substitution is performed by asking to substitute a faulty candidate service with a new one.

The self-healing and mediation modules have been tested also within the WS-Diamond Project, where a repair plan is executed according to the actions determined by a *Diagnoser* and a *Plan Generator*.

4 Scenarios and applicability criteria

XXX DA sistemare

The PAWS framework has been implemented in Java, using AXIS handlers to realize the mediation engine, extending jUDDI to provide service retrieval functionalities, using WordNet as a semantic network for term-based similarity evaluations, ActiveBPEL as a BPEL engine, extended with plug-ins for managing repair actions, WSDM to provide notification services to support process management operations, and CPLEX as optimization engine. All PAWS modules are implemented as services themselves, and can be used in different combinations to support flexibility and adaptivity. PAWS modules are available as open-source code on request.

The PAWS framework has been widely tested in multiple contexts to realize adaptive information systems based on Web services. In the MAIS project (http://www.mais-project.it), the proof of concept prototypes are a Virtual Travel Agency which allows selling travel packages by accessing the system from multiple channels and a micro MAIS application to gather on-field information with PDAs connected through a mobile network on archeological sites after a natural disaster. WS-DIAMOND (http://wsdiamond.di.unito.it) focuses on the development of service based self-healing systems. The reference scenario is a food selling company which cooperates with several

partners to sell food packages. Easylog (http://www.easylog.org/) goal is to support the risk management in dangerous goods transportation. Finally, the DISCoRSO (http://www.discorso.eng.it) project aims at developing flexible processes to support Small and Medium Enterprises.

XXX Sistemare In general, all developed applications require a basic adaptivity layer composed of the combination of URBE services to select candidate services, and the Mediation Engine to be able to adapt service automatically, transforming input and output messages and operation names to adapt service requests to the interface of invoked services. The Mediation Configurator is needed under the open world assumption, when candidate services are gathered from open UDDI registries; in fact, in this case the mapping between service interfaces may require manual intervention. The Process Optimizer has been adopted in cases where QoS global constraints are assumed to be critical, while SLA generation has been used for cases in which the QoS is provided as a range of possible values. In general, SLA generation and self-healing functionalities, introduced in the more recent WS-Diamond project, are required when a monitored and fault tolerant environment must be provided.

Pro: QoS non si/no ma agreement con ottimalita sia proc che lato provider Pareto ottimalita Contro: Performance. Applicabilita sul campo. Server su dispositivi piu risorse power consumption

5 Conclusions and Future Work

Further enhancements have been already planned with the intention to extend the PAWS framework along several directions.

For what concerns self-configuring, we aim at extending the mediation capabilities to reduce the design time effort and to automate mediation configuration as much as possible, dealing also with complex data structures. Moreover, we aim at strengthening adaptivity by introducing new and more flexible context management mechanisms to express user's and provider's preferences on the QoS of the composed process. Besides working on the mediation layer, we also aim at en-

hancing adaptivity by extending negotiation functionalities through the description of the contracts between users and providers.

Also the self-optimizing aspect of the PAWS framework will be extended, by taking into account the probability of faults, and reducing the optimization overhead by considering the optimization of multiple process instances. We aim at enhancing self-healability by introducing planning capabilities so that an orchestrated process can be recovered not only by using atomic recovery actions but also through a combination of them (XXX rif a WS-D).

In an orthogonal position with respect to the self-* properties of autonomic systems, there is the issue concerning the service description, and in particular we will investigate how to extend Web service descriptions in order to enhance the retrieval capabilities of the service registry.

More details on PAWS modules can be found in previous papers, in particular [**?**] for advanced service retrieval from registries, [**?**] for process optimization, [**?**, **?**] for QoS negotiation, [**?**] for dynamic substitution and message transformation algorithms, and in the Web site [**?**].

Acknowledgments

The work reported in this paper has been partially supported by the EU Commission within the IST FET-STREP Project WS-Diamond, by the MIUR MAIS FIRB Project, and by DISCoRSO, Easylog, and Quadrantis FAR Projects.

XXX Aggiungere CACM Papazoglu

Authors' biographies

- **Danilo Ardagna** is an assistant professor at Politecnico di Milano. His research interests include services composition, autonomic computing, and computer system costs minimization.
- **Marco Comuzzi** is a research associate at Politecnico di Milano. His research interests concern automated negotiation algorithms applied in the context of Web service quality.
- **Enrico Mussi** is a research associate at Politecnico di Milano. His research interests include context-based and self-healing compositions of Web services and service mediation.
- **Barbara Pernici** is full professor of Computer Engineering at Politecnico di Milano. Her

research activity is in the area of information and service engineering. She is chair of IFIP TC8 Information Systems and of WG 8.1 on Information Systems Design.

- **Pierluigi Plebani** is a research associate at Politecnico di Milano. His research interests concern Web service retrieval methods driven by both functional and quality aspects.