Cross-layer Self-adaptation of Service-oriented Architectures

Eli Gjørven
Simula Research Laboratory
P.O.Box 134
1325 Lysaker, Norway
eligj@simula.no

Romain Rouvoy
University of Oslo
Department of Informatics
P.O.Box 1080 Blindern
0316 Oslo, Norway
rouvoy@ifi.uio.no

Frank Eliassen
University of Oslo
Department of Informatics
P.O.Box 1080 Blindern
0316 Oslo, Norway
frank@ifi.uio.no

ABSTRACT

Service-Oriented Architectures (SOA) are built as compositions of inter-organizational services. These services are deployed and published by providers who are responsible for provisioning the services with sufficient resources. However, even though services are implemented by technologies providing a wide range of adaptation related features, such as configurable component models and communication infrastructures, state-of-the-art approaches to adaptive SOA systems do not provide principled solutions to exploit application layer adaptation mechanisms.

Our technology-agnostic adaptation middleware has been designed for integrating and exploiting technology-specific adaptation technologies and mechanisms. In this paper, we describe how this middleware can support a cross-layer adaptation of SOA systems. In particular, we focus on the server-side perspective of SOA, and we show that our middleware is able to exploit both service interface and application layers technologies for supporting a coordinated adaptation of both layers.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Software configuration management; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design

Keywords

Service-Oriented Architectures, Planning-based Adaptation Framework

1. INTRODUCTION

One of the current challenges for building self-adaptive systems is to cope with the increasing heterogeneity of applications and services, thus integrating multiple systems implemented in different platform and language technologies [9]. In this domain, interoperability issues are now achieved by open standards. However, other cross-cutting concerns, including adaptation, still lack solutions ensuring the coordinated reconfiguration of services developed with heterogeneous technologies. As self-adaptive systems relies on adaptation mechanisms, technology integration in this context also requires the *exploitation of adaptation features*, meaning that the system is able to exploit adaptation mechanisms provided by the platforms and languages available in the system. The definition of principled solutions to adaptation feature exploitation has not received much attention yet.

As a matter of example, *Service-Oriented Architectures* (SOA) has been identified as a principled approach to build systems from compositions of inter-organizational services. These services, generally developed using different technologies, are configured and published by providers who are responsible for provisioning the services with sufficient resources. As described by Erl [7], application logic in the context of SOA, can be split into two layers: the *service interface layer*, where loosely coupled services, hiding their implementation and technology platform, communicate via open protocols, and the *application layer*, in which service application logic is developed and deployed on different technology platforms.

In order to be able to adapt efficiently to run-time contextual changes, SOA-based systems must be able to adapt at the layer, or layers, that provides the best system improvement. This requires an adaptation framework that is able to integrate and control adaptation mechanisms in the two layers, without extensive re-factoring and re-implementation. However, current approaches to self-adaptation in the SOA community mostly focus on adapting the service interface layer, by exploiting the loose coupling between clients and servers in order to provide adaptation through service specification, selection, composition, and coordination mechanisms [8, 10, 13, 15, 16, 17]. They do not provide principled solutions for integrating state-of-the-art research in application layer self-adaptation, which has provided a wide range of mechanisms for adapting also the application layer. For example component models, such as FRACTAL [3] and OPENCOM [4], support dynamic configuration and reconfiguration of component-based applications, and middleware mechanisms supporting self-adaptation have been built from these component models [1, 5]. Thus, bridging SOA self-adaptation with application layer mechanisms, while preserving SOA principles as loose coupling, autonomy, and implementation and platform encapsulation, becomes a key challenge.

In our earlier research, we have developed an adaptation framework, QUA, which is technology-agnostic and able to integrate and exploit different technologies providing adaptation related mechanisms [9]. In this paper we show how this adaptation framework can be applied to SOA-based systems to perform *cross-layer adaptation*. By cross-layer adaptation, we generally mean adaptation of a system consisting of several layers, where the technologies and mechanisms of each layer are integrated and controlled by the same adaptation framework. Particularly in the context of SOA it means coherent adaptation across the service interface and application layers of a SOA system while preserving the loose coupling and autonomy of the services. QUA can also be applied to these two layers separately. This approach enables the extension of existing SOA-based self-adaptive mechanisms with a principled approach to adaptation management and control. Being technology-agnostic, QUA can also integrate legacy systems into a SOA system.

In this paper, we focus on how QUA can be applied to the server-side perspective in order to facilitate SOA implementation technology integration and exploitation, including: *i)* the *integration* of different types of service and work-flow specification languages, and *ii)* the *exploitation* of service implementation technology-specific features when available. The resulting system is able to perform cross-layer adaptation exploiting mechanisms both in the service interface layer, and the application layer in a coordinated fashion.

In the remaining of the paper, we first survey some related work in section 2. In section 3, we describe the design of a technology agnostic adaptation middleware, QUA, and in section 4 we demonstrate how the QUA middleware can integrate and exploit adaptation mechanisms of SOA systems. In section 5 we evaluate our solution by an example, and we conclude the paper in section 6.

2. RELATED WORK

As SOA applications are built from composing loosely coupled, potentially autonomous, services into work-flows, current approaches to self-adaptive SOA-based systems focus on the service interface layer. First, there are several standardization efforts that propose to extend the expressiveness of the *Web Service Definition Language* (WSDL) with support for defining extra-functional properties. For example, the WS-Policy standard [10] can be used to specify policies expressing non-functional requirements for *web services* (WS), while [16] proposes a language, WS-QoS, for specifying provided and required web services QoS. Second, in the area of service composition, several projects [8, 17] attempt to improve the dependability of SOA systems by applying adaptation frameworks that support dynamic web service selection and composition, based on web service process execution languages, such as the *Business Process Execution Language* (BPEL) [13]. Finally, these solutions can be combined, such as described by [15], where coordinated web service processes is composed from service composition, coordination, and policies. While these solutions provide a rich set of adaptation mechanisms for the service interface layer, we are not aware of solutions that provide principled approaches for integrating existing solutions to application layer adaptation into the service interface layer adaptation framework.

In self-adaptation middleware research, the current trend is to isolate the adaptation concerns from the application logic using generic *adaptation frameworks* [1, 5]. The adaptation framework is responsible for controlling the ongoing adaptation processes by constantly observing and analyzing the behavior of the target adaptive system, and by planning, instantiating, and executing adaptations when necessary. The adaptation framework depends on *adaptation mechanisms* that perform adaptation related tasks such as context monitoring, component life-cycle handling and reconfiguration, and adaptation of communication infrastructures [3, 4], and on *adaptation policies* used to decide which adaptation to carry out in each situation. Current adaptation frameworks are designed to adapt different parts of distributed systems, such as application software, middleware infrastructure (*e.g.*, communication, transaction, persistence), lower level operating system modules (*e.g.*, scheduler, driver), or device resources (*e.g.*, screen resolution, network interface). However, as discussed in [9], these adaptation frameworks are often tightly coupled to a fixed set of adaptation mechanisms and adaptation target technologies, and does not facilitate technology integration. For example, adaptation frameworks designed for component based systems [1, 5] provide adaptation primitives that operate on component implementations, like instantiation, configuration, and binding. This type of primitives does not suit the above-mentioned characteristics of loose coupling and autonomy of SOA-based systems.

3. THE QUA MIDDLEWARE

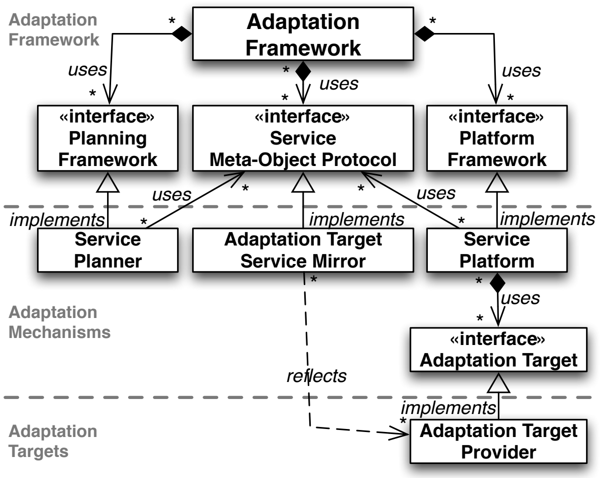
Adaptation Framework
Adaptation Framework
uses
uses
uses
«interface» Planning Framework
«interface» Service Meta-Object Protocol
«interface» Platform Framework
uses
uses
implements
implements
implements
Service Planner
Adaptation Target Service Mirror
Service Platform
Adaptation Mechanisms
uses
«interface» Adaptation Target
reflects
implements
Adaptation Targets
Adaptation Target Provider

Figure 1: The QUA adaptation middleware.

The QUA adaptation middleware has been designed with the goal of being technology-agnostic, providing a clean separation between the adaptation framework, the adaptation mechanisms, and the adaptation target. As can be seen in figure 1, QUA provides a clean separation between the adaptation framework layer, the adaptation mechanisms layer, and the adaptation target layer. As can be seen, the QUA adaptation framework consists of a *planning framework*, which is responsible for selecting service implementations and configurations, a *platform framework*, which encapsulates mechanisms for managing and adapting services, and a supporting *Service Meta-Object Protocol* (Service MOP). As we discussed in [9], the separation between the adaptation framework layer and the mechanisms layer facilitates building powerful adaptation middleware by integrating and exploiting different adaptation technologies. In particular, as we show later in this paper, this separation facilitates the implementation of cross-layer adaptive systems.

3.1 Service Meta-object Protocol

At the core of the QUA middleware is the service MOP mentioned above, providing service meta-information to the planning and platform frameworks through a reflective API. The service MOP is based on *mirror reflection*, where reflection is provided by separate meta-objects providing access to meta-information and -operation, instead of by the reflected system [2]. Different from reflective APIs provided by state-of-the-art adaptive middleware like Fractal [3] and OpenCom [4], the QUA service MOP abstracts all technology specific meta-information, such as roles and interfaces specific to particular component models, binding models, and configuration mechanisms. The service meta-model defines only a minimal set of technology agnostic concepts required by the planning and platform frameworks, as described below.

In order to hide the details of particular technology specific deployment artefacts, service implementations are deployed to the middleware as *implementation blueprints*, which are binaries encapsulating implementation artefacts like implementation classes, component or service descriptors, and configuration scripts. Any such blueprint is associated with a *service platform*, which encapsulates the run-time environment and technology-specific mechanisms required to interpret the blueprint. For example, such as Java classes depend on a Java Virtual Machine to instantiate and execute objects, and as software components depend on component containers in order to instantiate components and execute configuration scripts. As a service implementation may depend on other services, for example as one component depends functionalities provided by other components, the meta-model must identify all *dependencies* of an implementation, The dependencies of an implementation can be specified by only naming their types, thus allowing alternative implementations for the type to exist.

In order to support adaptation reasoning algorithms and policies that are not technology specific, in terms of system architecture, deployment environment, or implementation classes, the service MOP reflects service quality on two abstraction levels. On the implementation level, a *quality predictor function*, or in short *predictor*, calculates the predicted quality provided by a blueprint in a certain execution context. A predictor function encodes technical knowledge about the implementation, such as component model or implementation architecture, and how its quality depends on execution context, such as resource availability and application context. Its output prediction is expressed by implementation technology independent quality dimensions like response time in milliseconds, or the expected availability in percentage of time. The dimensions used to specify the quality of a service, is defined by the type of service. On the user level, a service client can express service quality requirements using *utility functions*, mapping quality prediction to a scalar value reflecting the usefulness of this service quality. Thus, the QuA meta-model isolates knowledge about implementation details in the quality predictors, and allows specification of technology agnostic adaptation policies through utility functions.

3.2 Planning Framework

The task of the planning framework, is to evaluate alternative implementations of services, and select the alternative that best satisfy the user requirements. The planning framework calculates the predicted quality of alternative implementations, and applies the utility function to select the alternative that provides the highest usefulness. As the predicted quality of a service may depend on the predicted quality of its dependencies, the service planner recursively finds alternative combinations of implementations and implementation dependencies, and calculates their complete quality. Due to the combination of predictor and utility functions, the planning algorithms are completely technology independent, and can thus be developed by algorithm experts, rather than adaptation or application domain experts. In [6], we describe a comprehensive application scenario demonstrating the application of the service meta-model, including examples of quality prediction and utility functions. The advantages of using utility functions in the context of self-adapting systems are further discussed in [11].

3.3 Platform Framework

The platform framework provides a plug-in mechanism for technology specific mechanisms that can be applied to service implementations. The platform framework is invoked by the middleware to execute a service implementation based on the blueprint selected by the planning framework. For example, if the planning platform selects a component implementation, the corresponding component platform is invoked with the selected blueprint. This platform must encapsulate the component container able to execute the component, including container specific mechanisms such as component life-cycle, parameterization, and binding frameworks, reconfiguration algorithms, state handling, and exception handling. The platform extracts component implementation classes, descriptors, and other component technology specific artefacts from the blueprint, and invokes the container to start the component. In [9], we describe in detail the design and implementation of a service platform integrating an advanced component model, including component model specific life-cycle, binding, and configuration mechanisms.

4. DESIGNING A CROSS-LAYER ADAPTATION MIDDLEWARE FOR SOA

In this section, we demonstrate how the QuA middleware can be applied in order to support self-adaptive SOA applications by integrating both interface layer and application layer mechanisms providing different degrees of adaptation. [1]

4.1 Self-adaptive SOA-based Systems

An adaptation framework supporting technology integration and exploitation in SOA-based systems must be able to exploit both application layer and service interface layer mechanisms. Furthermore, to respect SOA principles such as interoperability, loose coupling, and autonomy, it must support the following requirements:

1. it must *not depend on concrete adaptation actions*—*i.e.*, component configuration, or service composition configuration—but rather on their technology independent abstractions like "configuration",
2. it must be *applicable to both loosely and tightly coupled systems*,
3. it must be able to *integrate adaptation related control interfaces* where they are available,
4. it must be able to *integrate autonomous services*, even if they do not provide adaptation related control interfaces.

In the following sections, we demonstrate that the QuA middleware is able satisfy these requirements.

4.2 Cross-layer Adaptation Middleware

Figure 2 illustrates the design of a cross-layer middleware based on the QuA adaptation framework. As indicated by the figure, the adaptive target system includes both the service interface layer and the application layer. As discussed in subsection 4.1, the adaptation mechanisms available in the two layers are quite different by nature. The integration of the two layers involves two tasks. [2]

First, the concepts and artefacts of the two layers must be mapped into the common QuA meta-model. For each available service implementation, this includes specifying its behavior, as a functional type, specifying its service platform, generating blueprints

[1]We emphasize that the integration described below integration of existing web services without making any modifications that compromise the execution of clients that are unaware of the QuA middleware.

[2]As indicated by the figure, the application layer may consist of subsystems implemented in different technologies. Note that the interoperability of the subsystems depends on the technologies. In the case of SOA-based systems, interoperability is ensured through satisfaction of the SOA principles.

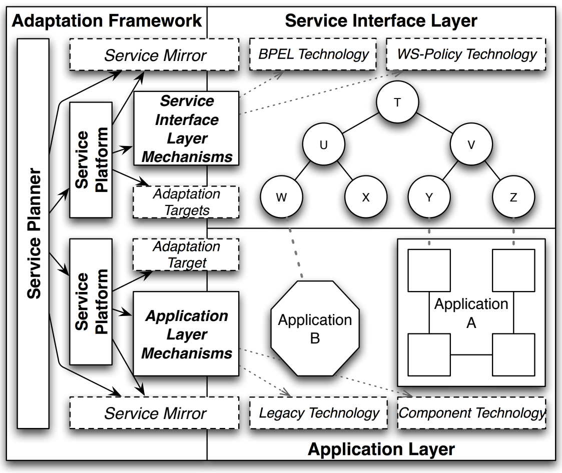

Figure 2: Design of the WS-PLANNER cross-layer adaptation middleware.

containing implementation code, deployment descriptors, configuration scripts etc., specifying its dependencies to other services, and writing quality predictions describing its qualitative behavior. From the resulting meta-information, the planning framework can derive the dependencies between the organization of the services in the service interface layer, and the implementation of the services in the application layer. Based on predictors and utility functions, all parts of the system can be considered together during planning, in order to select an optimal combination of service orchestration and application implementation.

Second, the technology specific platforms must be developed, each encapsulating the run-time environment and adaptation mechanisms provided by its technology. While application layer platforms make application layer implementations available for remote connection, service interface layer platforms integrate these remote connection points as web service end-points during web service coordination and orchestration. Each platform instantiates and adapts services based on information extracted from the blueprint selected by the planning framework. Note that even if application layer adaptation mechanisms are supported by application technologies, they are not always available to the QUA middleware. For example, configuration interfaces of application components executing in an external administration domain will normally not be available. The ability to influence service behavior would in this case depend on the support for service specification standards and SLA negotiations on the service interface layer.

4.3 Integration of Service Interface Layer Adaptations

4.3.1 *Supporting Web Services Interfaces*

The first approach is a simple integration inspired by [14], where the middleware connects to web service end-points identified by WSDL documents, and returns proxies to these end-points. The WS platform interprets *WS blueprints*, encapsulating a WSDL document containing service address and protocol information. The WSDL document is extracted from the blueprint, and the web service programming API is invoked in order to connect to the web service end-point as a client, using any web service programming interface. The WS platform is able to connect to any web service satisfying the WSDL specification, but it does not support advanced web service technologies, nor any kind of adaptation.

In order to enable the QUA adaptation framework to evaluate the quality provided by a web service, *WS quality predictors* can be developed and added manually using the QUA reflective API.

4.3.2 *Supporting Web Services Policies*

In the second approach, we extend the first approach with support for specification of web services policies using the policy language WS-Policy [10]. A service policy can be used to express requirements, or capabilities, of a required, or provided, web service. From the server perspective, by supporting WS-policies, a service provider allows a service client to specify requirements for non-functional behaviors. WS-Policy defines three operators that all operate on a list of non-functional behaviors $(x_1, .., x_n)$. $ExactlyOne(x_1, .., x_n)$ means that exactly one of $(x_1, .., x_n)$ must be supported. $OneOrMore(x_1, .., x_n)$ means that one or more of $(x_1, .., x_n)$ must be supported. $All(x_1, .., x_n)$ means that all of $(x_1, .., x_n)$ must be supported.

WS-policies are suitable to specify behaviors that are closely related to implementation variants, as demonstrated by the algorithm selection policy in the example given below. Thus, we extend the solution presented in subsubsection 4.3.1 as follows. As the WS platform described above, a *WS-Policy platform* is able to connect to a web service end-point identified by a WSDL document, and return a proxy to the resulting end-point. However, the WS-Policy platform also intercepts calls made to the proxy, and attaches policy specifications to the messages. A WS-Policy platform interprets a *WS-Policy blueprint* containing WSDL specifications and WS-Policy specifications. *WS-Policy quality predictors* that reflects the qualitative properties of each valid policy alternative, can be defined in order to enable the planning middleware to select which policy to apply based on utility functions.

We demonstrate this mapping by the following example. A message protocol supports two security extensions; digital signature ($sign$) and shared key encryption ($encrypt$). A security policy can be applied to the protocol to decide which mechanisms should be used. We can deploy this protocol as a WS-Policy platform service. Each valid protocol alternative is represented by a WS-Policy blueprint encapsulating the service WSDL, and a legal security policy to be read and invoked by the WS-Policy platform.

We define a quality model for this message protocol with four quality dimensions: Authentication (A), integrity (I), non-repudiation (R), and confidentiality (C). Predictor functions (P) can be used to map between the security policies that can be applied to the protocol, and these dimensions:

- $P(All(sign, encrypt)) = \{A, I, R, C\}$
- $P(ExactlyOne(sign)) = \{A, I, R\}$
- $P(ExactlyOne(encrypt)) = \{C, I\}$

Policies applying the $OneOrMore$ operator is not included here. Asking for $OneOrMore(sign, encrypt)$ gives a random security level, which does not make sense. When specifying service requirements for this service, a utility function must be provided, that reflects the security requirements of the user by weighting the dimensions towards each other.

4.3.3 *Supporting Web Services Compositions*

In the third approach, we integrate support for web services composition, using a language for business process orchestration, called *Business Process Execution Language* (BPEL) [13]. A BPEL process defines a sequence of activities and constraints for their execution. The process also defines relationships to external partners, also called *partner links*, that provide or require services that are invoked during the execution of the process. A BPEL process itself, ultimately provides a service. Thus, we consider a BPEL process description as a potential service implementation that can be functionally and qualitatively described, planned and instantiated by a QuA platform. Furthermore, we may represent BPEL partner links explicitly as service dependencies, which can be planned, instantiated and adapted independently by the QuA middleware.

In this respect, a *WS-BPEL platform* interprets *WS-BPEL blueprints*, encapsulating BPEL scripts, and invokes a BPEL engine to execute the scripts. In the case of explicitly defined dependencies for partner link, these services are specified as abstract services in the BPEL script. When the middleware has obtained references to concrete services implementing the partner link dependencies, they are presented to the WS-BPEL platform as service end-points. Based on meta-data enclosed in the blueprint, the WS-BPEL platform injects into the abstract references in the BPEL script, the concrete references to corresponding service end-points, identifying their protocol and address information. The resulting script is a complete script that can be directly executed by the BPEL engine.

As a simple illustration, we describe the problem of travel planning as a BPEL process. A travel agency provides travel planning as a process consists of three activities; Through *Travel Exploration*, the customer can view different types of material (pictures, videos, etc.) describing alternative places to travel. Then, the customer proceeds to *hotel and flights booking*, and finally, everything is *payed in one transaction*. Each of these activities refers to a partner link service, which can be defined as an implementation dependency. The QuA middleware can be applied recursively, in order to plan, instantiate, and adapt the composed travel planning service, implemented by the aforementioned BPEL script, and its partner link services.

As the quality of a BPEL process depends on the quality of its partner link services, a *WS-BPEL quality predictor* predicts the quality of a BPEL process based on predictions of the concrete services implementing its partner link services. Thus, during planning, the quality of the BPEL process is calculated based on combinations of alternative partner link implementations. In the case of travel planning, the security level of the entire process depends on the security level provided by the payment service and the reservation services. The quality predictor functions must reflect these dependencies, and the user level security requirements can be reflected by utility functions in similar terms as described in subsubsection 4.3.2. In section 5, we discuss this example in detail.

4.4 Integration of Application Layer Adaptations

4.4.1 Supporting Component-based Reconfiguration

The Travel Exploration service described above, can be implemented by software supporting different types of adaptation, such as configurable component models and adaptive middleware. In [9], we demonstrated how such an adaptive middleware can be integrated into the QuA middleware. In order to integrate the FRACTAL component model [3], and adaptation mechanisms based on this technology, we developed a *Fractal platform* that was able to instantiate and adapt application layer FRACTAL components to the currently available resources.

The Fractal platform depends on an implementation of the Fractal component runtime, an *Architecture Description Language* (ADL) called FRACTALADL [12], used to deploy FRACTAL applications, and a component reconfiguration script language and interpreter called FSCRIPT [5]. Implementation classes, ADL descriptors and reconfiguration scripts were deployed to the FRACTAL platform as *Fractal blueprints*, and they were accompanied by *Fractal quality predictors* describing their service level. An exhaustive description of this experiment can be found in [9]. This technology can be extended to support adaptation of applications, such as the Travel Exploration service provided by the travel agency.

4.4.2 Supporting Legacy Applications

As the QuA middleware does not require integrated services to be implemented by any particular technology. Integration of legacy applications may have to be designed using application-specific adaptation mechanisms. In [6], we applied the QuA middleware to a state-of-the-art adaptive video streaming application that was not based on an advanced component technology, but rather implemented using so-called POJOs - *Plain Old Java Objects*. The application was ported to the middleware with rather small efforts. Thus, we expect that such legacy applications also can be integrated with the SOA-based approaches presented above.

5. EVALUATION

In order to evaluate the solution described in this paper, we illustrate in more details how cross-layer adaptation is achieved by applying the QuA middleware to the example technologies described in section 4. Finally, we evaluate the solution by applying the requirements presented in subsection 4.1.

5.1 Example: Travel Planning Scenario

The travel planning scenario is an example of a system where cross-layer adaptation can be applied in order to better exploit the adaptation potential provided by application and service interface layer technologies. We now take a closer look at how the middleware proceeds to plan, instantiate and adapt the composed travel planning service.

Initially, when the QuA middleware is invoked in order to start the travel planning application, the planning framework locates the BPEL script we described in subsubsection 4.3.3. Three dependencies are identified: *i)* the *travel exploration* service, *ii) hotel and flight reservation* services, and *iii) payment* service. The planning framework locates alternative implementations for each service, calculates the predicted quality for the alternative dependency resolutions, and selects the alternative with the highest utility.

In this example, partner service implementations are selected as follows. The *travel exploration* service is a local server provided by the travel agency. It has been implemented using the FRACTAL component model as described in subsection 4.4. Hence, the actual service implementation can be dynamically adapted to context information, such as the number of clients, and the data they are downloading. The *hotel and flight reservation* services are web services as described in subsubsection 4.3.1 and 4.3.2, provided by third party service providers. Such reservation services may provide the possibility of applying web service policies, but a part from that, the services are not configurable. The *payment* service is a web service provided by the financial department of the travel agency. Considering the strict security requirements of financial transactions, these services typically have fixed functional and non-functional requirements—*i.e.*, security policies—that can not be altered.

When the complete implementation has been selected, the middleware starts instantiating the composed service in a bottom-up fashion, starting with the partner link services. Each partner link service platform is invoked to instantiate the implementation encapsulated by the selected blueprint. The way the partner link services are instantiated and configured, depends on the level of control facilitated by the partner link service provider. The *hotel and flight reservation* service partner link services are existing web services residing in external administration domains. Thus, their implementation cannot be inspected and manipulated in any way, they can only be accessed through their specified web service end-point. In such cases, instantiation does not include the actual creation of instances, but rather connecting to the external service instances, and return proxy references to it. On the other hand, the local *travel exploration* service provides FRACTAL control interfaces that can be exploited for component instantiation and configuration by the FRACTAL platform.

Finally, the middleware can initiate adaptation of the application during the execution of the services. If new implementations are selected, reconfiguration is performed by the service platforms. For example, web services that becomes unavailable can be replaced, and FRACTAL components that are not performing good enough, can be re-planned by the planning framework, and reconfigured by the FRACTAL platform.

5.2 Assessment of the Requirements

In subsection 4.1, we presented four requirements that cross-layer adaptation should discuss. When applying these requirements to the proposed solution, we conclude that:

1. *It does not depend on concrete adaptation actions*: actions are encapsulated by the platform framework and technology-specific information in the blueprints (cf. subsection 4.2);
2. *It is applicable to both loosely and tightly coupled systems*—*i.e.*, both loose coupling to web services end-points (cf. subsubsection 4.3.3) and tightly coupled components (cf. subsection 4.4);
3. *It is able to integrate adaptation related control interfaces* where they are available—*i.e.*, FRACTAL controllers (cf. subsection 4.4);
4. *It is able to integrate autonomous services*—*i.e.*, web services end-points as in subsubsection 4.3.1.

Through the planning and platform framework, and the common service meta-model, mechanisms from the service interface layer and the application layer can be integrated, in order to bridge the two layers, while preserving the SOA principles. Thus, we ensure that web services managed by the QUA middleware can still be accessed by any web service client, also clients that are unaware of the QUA middleware.

Even though WS-Policy, BPEL, and similar languages have been used by several projects to provide adaptive web service coordination and orchestration [8, 15, 18], we are not aware of any projects that has been able to provide principled solution to combine such languages with application level adaptation mechanisms. The middleware is able not only to integrate services with different degrees of available adaptation support, but also to exploit the adaptation mechanisms when available. As demonstrated by the examples, the technology-agnostic planning and platform frameworks and meta-model, enables the WS-PLANNER middleware to recursively exploit technology specific adaptation mechanisms to both the service interface layer and the application layer, without compromising the principles of loose coupling, autonomy, and implementation and platform encapsulation.

6. CONCLUSIONS AND FUTURE WORK

In this paper we described an adaptation middleware that is able to support the integration and exploitation of various adaptation technologies, including SOA-based adaptation mechanisms such as service selection and composition. This middleware separates between the adaptation framework, providing technology independent adaptation reasoning and adaptation strategies, and adaptation mechanisms, which are technology specific. In the context of SOA-based applications, we demonstrated that the middleware can be applied to achieve cross-layer adaptation that performs adaptation on the service interface layer and the application layer in an integrated fashion, while preserving the loose coupling and autonomy of services. In future work, we plan to extend this work with support for *Service Level Agreement* (SLA) negotiation.

7. REFERENCES

[1] T. V. Batista, A. Joolia, and G. Coulson. Managing Dynamic Reconfiguration in Component-Based Systems. In *2nd European Workshop on Software Architecture (EWSA)*, volume 3527 of *LNCS*, pages 1–17. Springer, June 2005.

[2] G. Bracha and D. Ungar. Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages. In *19th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 331–344. ACM, 2004.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in Java. *Software Practice and Experience (SPE)*, 36(11/12):1257–1284, Aug. 2006.

[4] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A Generic Component Model for Building Systems Software. *ACM Transactions on Computer Systems (TOCS)*, 26(1):1–42, 2008.

[5] P.-C. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive FRACTAL Components. In *5th International Symposium on Software Composition (SC)*, volume 4089 of *LNCS*, pages 82–97. Springer, Mar. 2006.

[6] F. Eliassen, E. Gjørven, V. S. W. Eide, and J. A. Michaelsen. Evolving self-adaptive services using planning-based reflective middleware. In *Proceedings of the 5th International Middleware Workshop on Adaptive and Reflective Middleware (ARM)*, volume 190, pages 1–6, Nov. 2006.

[7] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[8] A. Erradi, P. Maheshwari, and V. Tosic. Policy-Driven Middleware for Self-adaptation of Web Services Compositions. In *7th International Middleware Conference*, volume 4290 of *LNCS*, pages 62–80. Springer, Nov. 2006.

[9] E. Gjørven, F. Eliassen, and R. Rouvoy. Experiences from Developing a Component Technology Agnostic Adaptation Framework. In *11th International Conference on Component-Based Software Engineering (CBSE)*, volume 5282 of *LNCS*. Springer, Oct. 2008.

[10] IBM. Web Services Policy Framework, Mar. 2006.

[11] J. O. Kephart and R. Das. Achieving Self-Management via Utility Functions. *IEEE Internet Computing*, 11(1):40–48, 2007.

[12] M. Leclercq, A. E. Özcan, V. Quéma, and J.-B. Stefani. Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In *29th International Conference on Software Engineering (ICSE)*, pages 209–219. IEEE, May 2007.

[13] OASIS. Web Services Business Process Execution Language, Apr. 2007.

[14] R. Rouvoy, F. Eliassen, J. Floch, S. Hallsteinsen, and E. Stav. Composing Components and Services using a Planning-based Adaptation Middleware. In *7th International Symposium on Software Composition (SC)*, volume 4954 of *LNCS*, pages 52–67. Springer, Mar. 2008.

[15] S. Tai, R. Khalaf, and T. A. Mikalsen. Composition of Coordinated Web Services. In *5th International Conference on Middleware*, volume 3231 of *LNCS*, pages 294–310. Springer, Oct. 2004.

[16] M. Tian, A. Gramm, H. Ritter, and J. H. Schiller. Efficient Selection and Monitoring of QoS-Aware Web Services with the WS-QoS Framework. In *International Conference on Web Intelligence (WI)*, pages 152–158. IEEE, Sept. 2004.

[17] T. Yu and K.-J. Lin. Adaptive algorithms for Finding Replacement Services in Autonomic Distributed Business Processes. In *7th International Symposium on Autonomous Decentralized Systems (ISADS)*, pages 427–433. IEEE, Apr. 2005.

[18] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.