Composing Components and Services Using a Planning-Based Adaptation Middleware

Romain Rouvoy[1], Frank Eliassen[1], Jacqueline Floch[2], Svein Hallsteinsen[2], and Erlend Stav[2]

[1] University of Oslo,
P.O. Box 1080 Blindern,
0316 Oslo, Norway
{rouvoy, frank}@ifi.uio.no
[2] SINTEF ICT,
7024 Trondheim, Norway
{jacqueline.floch, svein.hallsteinsen, erlend.stav}@sintef.no

Abstract. Self-adaptive component-based architectures provide methods and mechanisms to support the dynamic adaptation of their structure under evolving execution context. Dynamic adaptation is particularly relevant in the domain of ubiquitous computing, which is subject to numerous unexpected changes of the execution context. In this paper, we focus on changes in the service provider landscape: business services may dynamically come and go, and their quality of service may vary. We introduce an extension of the MADAM component-based planning framework that optimizes the overall utility of applications when such changes occur. MADAM planning is based on dynamic configuration of component frameworks. The extended planning framework supports seamless configuration of component frameworks based on both local and remote components and services. In particular, components and services can be plugged in interchangeably to provide functionalities defined by the component framework. The extended planning framework is illustrated and validated on a use case scenario.

Keywords: Adaptation planning, component-based architectures, self-adaptation, service-oriented architectures.

1 Introduction

Self-adaptive architectures provide methods and mechanisms supporting the dynamic adaptation of their structure under an evolving runtime execution context. Dynamic adaptation is particularly relevant in the domain of ubiquitous computing, where users carrying mobile devices move around in ubiquitous computing environments causing frequent and unexpected changes in the execution context of their applications. For example, a mobile device is frequently roaming, and its applications have to be dynamically adapted to remain useful under new network conditions. Such an adaptation requires the detection of context changes, but also the selection of an

C. Pautasso and É. Tanter (Eds.): SC 2008, LNCS 4954, pp. 52–67, 2008.

application configuration that maintains a satisfactory *Quality of Service* (QoS) in the new context. With the emergence of *Service-Oriented Architectures* (SOA) [1], both the availability and the QoS of the services on which the applications depend become an important part of the context. Thus, SOA is of interest for self-adaptive applications, because services are reusable and composable entities that can be dynamically exploited to improve the behavior of an application executed on a mobile device. Services in a SOA environment can be discovered and accessed without knowledge of the underlying platform implementation and hence can be exploited in the dynamic configuration of the applications. Adaptation in MADAM is generally QoS-driven. In SOA, QoS properties are part of the *Service Level Agreements* (SLAs) [2] that are negotiated between a service provider and its end-user consumers. By integrating SLA negotiation into the adaptation decision process, application adaptation exploiting SOA can still be QoS-driven.

The MUSIC planning framework introduced in this paper is an extension of the MADAM planning framework, which supports the adaptation of component-based architectures [3]. MADAM follows an architecture-centric approach where we represent architecture models at runtime to allow generic middleware components to reason about and control adaptation aims at simplifying the development of adaptive applications. In MADAM applications are modeled as component frameworks where functionality defined by a component framework can be dynamically configured with conforming component implementations. The purpose of an adaptation-planning framework is to compute and evaluate the utility of alternative configurations in response to context changes, and to select a good one for the current context. The extension we propose supports self-adaptation of ubiquitous applications to changes in the service provider landscape. The planning middleware evaluates discovered remote services as alternative configurations for the functionalities required by an application. This means that the extended planning framework, when triggering an adaptation, can support seamless configuration of component frameworks based on both local and remote components and services. In particular, components and services can be plugged in interchangeably to provide implementation of functionalities defined by the component framework. In the case of services, the planning framework deals directly with SLA protocols supported by the service providers to negotiate the appropriate QoS for the user.

In this paper, we first introduce a motivating scenario for the support of remote services in mobile applications (cf. Section 2). After presenting the various foundations of this work (cf. Section 3), we introduce our planning framework capable of supporting SOA when adapting applications (cf. Section 4). This planning framework is illustrated and validated on a use case extracted from the motivating scenario (cf. Section 5). Finally, related work is discussed (cf. Section 6) before concluding and presenting our perspectives (cf. Section 7).

2 Motivating Scenario

To further motivate the need for adaptation in service-oriented computing environments, let us consider the following scenario. A sales agent spends much of his time visiting customers. To assist him in his work, he is using an extended

Customer Relationship Management (CRM) system. The system offers traditional CRM functionality, such as keeping track of and sharing customer- and business-related information. In addition, it assists the agent with route planning, detection of traveling delays and notifying customers affected by such delays.

Scene 1. The scenario starts with the sales agent meeting a customer, and using the CRM system on his laptop to record agreements with the customer. Before the meeting terminates, he is notified about an upcoming meeting at another customer site and decides to prepare for this new meeting. He picks up his smartphone and launches the mobile CRM application to find the best route and estimate the travel time. For this, the application uses a location service, a map service, and a route planning service. There are several providers both for the map service and the route planning service, and the CRM application has to select providers facilitating a quick and precise answer to the agent. In this case the location service provided by the WLAN at the customer site and a route planning service available through the Internet are the best alternatives.

Scene 2. The agent ends the current meeting, and walks out to his car to go to his next meeting. As he drives away from the customer's building, the smartphone looses connection to the customer WLAN, and the Internet connection is switched seamlessly to GPRS by the mobile IP software installed on the smartphone. The connection to the location service, which the CRM application needs to monitor progress, is lost. The GPRS provider also offers a location service, but with a lower accuracy. However, the car has a navigation system based on GPS, which provides a more accurate location service via a Bluetooth connection. The car navigation system also offers a navigation aid service. The CRM application reconfigures itself to use these services, since this solution provides a better accuracy, and a increased visibility to the user because of the larger display of the car navigation system. It also saves battery life on the smartphone since the navigation aid component of the CRM application has been replaced by an external service.

Scene 3. Half way to the meeting, the agent runs into a traffic jam caused by an accident partly blocking the road, resulting in a temporary slowdown of his progress towards his next destination. The CRM application detects this situation, alerts the agent that he will be late, estimates the delay using data obtained from the route planning service and offers the agent to notify affected customers. The agent accepts this proposal and the CRM application sends text messages to the customers using the available smartphone interface. The CRM application monitors progress and re-estimates the arrival time regularly in order to be able to alert the meeting about changes. Meanwhile the selected route planning service becomes congested, leading to slow response and out-of-date information. The application detects this and reconfigures to an alternative service that costs more to use but which provides more up-to-date information.

3 Foundations

This section introduces the basis of the proposed approach by presenting concepts related to planning-based middleware (cf. Section 3.1), and service-oriented

architectures (cf. Section 3.2). The section concludes by identifying the assumptions that are considered in our contribution (cf. Section 3.3).

3.1 Planning-Based Middleware

Planning-based middleware refers to the capability of adapting an application to changing operating conditions by exploiting knowledge about its composition and *Quality of Service* (QoS) metadata associated to the application components [4]. We therefore consider applications that are developed with a QoS-aware component model. The QoS model associated with a ubiquitous application defines all the reasoning dimensions used by the planning-based middleware to select and deploy the component implementations that contribute to provide the best utility. The utility of an application grows when its constituting components better fulfill user preferences while optimizing device resource consumption.

Planning refers to the process of selecting components that make up an application variant that provides the best possible utility to the end user. This process can be triggered during several steps of the application life cycle, such as during the deployment of the application or at runtime if the execution context suddenly changes. The parts of the application that are considered during planning are called *variation points*. These correspond to functionalities (type of behavior) defined in the component frameworks modeling the application. Thus, each variation point identifies a functionality of the application that can be implemented differently. In addition, each component implementation suitable for a variation point is reified as a *plan* by the planning-based middleware. A plan mainly consists of a structure that reflects the type of the component implementation and the QoS properties associated to the services it provides. In particular, the plan exhibits both *requested properties* (*e.g.*, memory consumption, network bandwidth, network connectivity) and *offered properties* (*e.g.*, request throughput, response time, result accuracy) referring to the QoS model of the application. To estimate the offered properties of a plan, the planning-based middleware relies on *property predictors*. The property predictors are used to predict the offered properties of a component implementation as a function using the required properties and the current execution context as parameters. The predictors can also take into account the state of the component implementation associated to the plan—*i.e.*, described, deployed, or running—to refine the prediction. The QoS model used by the planning framework can be customized to handle new QoS dimensions (*e.g.*, monetary cost), while the property predictors can be configured to support complex heuristics (*e.g.*, QoS negotiation protocols). The predicted properties are input to a normalized *utility function* that computes the expected utility of a composition of plans making up an alternative application configuration. The planning-based middleware compares the expected utility of all alternate application configurations, and finally selects the one that provides the highest value.

Fig. 1 illustrates the architecture of the MADAM adaptation middleware. The component Adaptation Manager supports the *planning procedure* by operating a generic reasoning heuristics that exploits metadata included in the available plans. In particular, the plans are composed based on their type compatibility to describe alternative application configurations. Then, the heuristics ranks the application configurations by evaluating their utility with regards to the application objectives.

This evaluation is achieved by computing the offered properties using the property predictors associated to each plan contained in the selected application configuration and retrieved from the component Plan Repository.

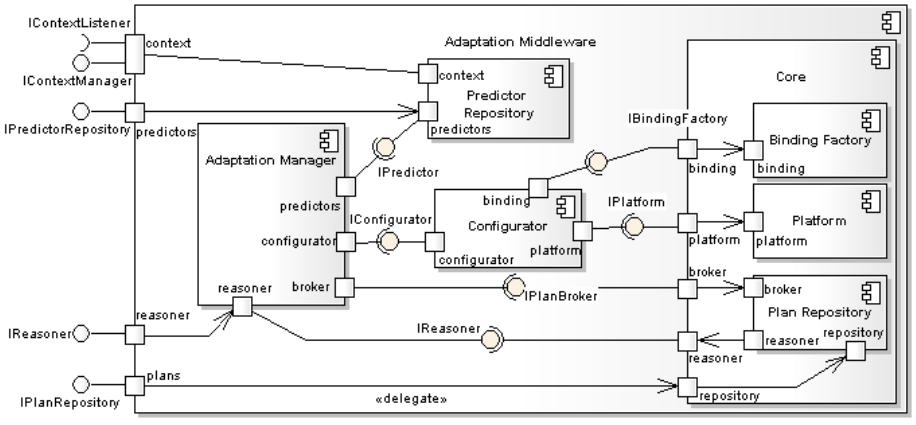

Fig. 1. The Architecture of the MADAM planning-based adaptation middleware

The component Plan Repository provides an interface IPlanBroker for the Adaptation Manager to retrieve plans associated with a given component type during planning. The Adaptation Manager may request plans that are compatible with a given variation point, at which point the Plan Repository will search for matching component types. Any additional metadata on the required component type will help the Plan Repository to exclude plans and filter the search space [5]. Plans are typically published to (and discarded from) the Plan Repository by applications and component development tools using the interface IPlanRepository, and can thus trigger the Adaptation Manager for re-planning of the application if needed (*e.g.*, the discarded plan was associated to a running component).

The *reconfiguration* process is handled by the component Configurator and consists of taking the set of plans selected by the component Adaptation Manager and reconfiguring the application. Before deploying the application configuration selected by the reasoning engine, the component Configurator brings the current application into a stable state, by suspending the execution of its contained components. Then, if the component described by a plan is in the running or deployed state, the associated component instance is configured for the variation point and connected to other components using the component Binding Factory. If the component is in the described state, then the component should be preliminary instantiated and deployed by the component Platform using the component implementation description associated to the plan. The result of the reconfiguration (*e.g.*, reference of the deployed instance) is automatically reflected into the selected plans.

Thus, the MADAM planning-based middleware offers a modular and extensible approach for adapting applications built with various types of component models. In particular, the concept of *plan* can be derived to support heterogeneous artifacts and

their associated states. Furthermore, the components Platform and Binding Factory provide sufficient abstractions for supporting different middleware platforms (*e.g.*, CORBA, J2EE, and Web Services).

3.2 Service-Oriented Architectures

When studying the SOA domain, we observe that there is no standard, universally accepted definition of *Service-Oriented Architectures*. Erl [1] proposes to characterize SOA by a set of fundamental design principles for service-orientation, such as abstraction, reusability, composition, and loose coupling. SOA can also be considered as an evolution of component-based architectures. In component-based software engineering, applications are assembled from components that can be used without any knowledge of either their implementation or their underlying platform. SOA goes a step further by introducing an abstract business model defining the concepts of *functionality* as a product or an enterprise resource, *service provider*, *service consumer,* and *service contract*. While the owner of a component-based application is responsible for the instantiation of components, the service provider is responsible for the creation and the management of services. The most fundamental principle of service-orientation is the standardized service contract [6]. In particular, services express their semantics and capabilities via a service contract. Although SOA was initially proposed to organize business software, service-orientation provides facilities that are applicable beyond that scope. For example, support has been developed for interface type and semantics descriptions, QoS descriptions, service discovery protocols, and binding factories. Nowadays, the SOA concepts are more and more exploited in a large set of producer/consumer systems, such as ubiquitous systems.

Service QoS properties are normally negotiated between the service provider and the service consumer, and are described as part of the service contract as a *Service Level Agreement* (SLA). A service level is used to describe the expected performance behavior, such as response time and availability, or other properties such as billing, termination terms and penalties in the case of violation of the SLA [1]. An SLA can simply be created after selection of a fixed service level offer among several pre-defined offers or, in more complex cases, after customization via a negotiation process. An SLA may be valid for a limited period, or may be terminated explicitly. During service provisioning, the provider should monitor the service quality, and adapt the resources to avoid a violation of an SLA. The consumer may also perform monitoring as well to avoid blindly trusting the provider.

In our work, support for services is motivated by the possibility to control the usability, usefulness, and reliability of a ubiquitous application by adapting it to changes in the service landscape. The following changes are relevant:

1. The service providers add (resp. remove) services in (resp. from) the environment,
2. New services become accessible depending on changes in the ubiquitous execution context, such as network conditions or locations,
3. The quality of service becomes better or worse due to context changes,
4. The violation of an SLA (by the user or the provider) leads to the termination of the SLA.

Mechanisms for discovering changes in the service landscape and contract violations are not discussed in this paper. The planning process is triggered when changes occur. Planning requires the ability to reason on service properties (including QoS) and dependencies between service properties and context.

3.3 Assumptions

Although it is also relevant to investigate our planning-based middleware to support the planning of service compositions, this paper concentrates on the adaptation of component-based applications operating in service-oriented environments. In particular, our middleware does not explicitly control the resources in the provider domain. Thus, we assume that the description and the deployment of SLA contracts, made available to customers, are realized by the service providers. We assume that service discovery and service levels identification are performed prior to planning. Whether a SLA contract or a set of potential SLA contracts are negotiated during discovery or during planning depends on the flexibility of provider offers and on the consumer needs. Thus, we aim at providing flexible solutions and foresee that a service level offered during service discovery may no longer be valid when requested after selection during planning. Our solution must therefore cope with the possible denial of the requested service level by the service provider.

4 Service Planning with SOA

Based on the above foundations, the SOA concepts can be integrated in our planning framework by supporting a common and uniform representation of the different forms of services—*i.e.,* component descriptions, component instances, and remote services. This common representation provides all the meta-information that is required to evaluate the utility of a service for a given application. Thus, when a service is discovered by the platform, its meta-information needs to be made available to the planning framework in a suitable form (cf. Section 4.1). And, if selected by the planning heuristic (cf. Section 4.2), a remote service should be connected to the ubiquitous application by using a proper binding framework that provides interoperability between the user- and provider-sides and that contributes to SLA monitoring (cf. Section 4.3).

4.1 Plan Discovery and Brokering

According to Fig. 2, we propose to extend the component Adaptation Middleware introduced in Fig. 1 to support the integration of remote services and service level agreements. In particular, we have introduced new components (the darkest ones in the figure) to support different types of remote services. To do so, the component Adaptation Middleware integrates a composite component SOA that isolates the integration of a given SOA technology (*e.g.*, Web Service, CORBA, or UPnP). This separation of concerns allows also the adaptation middleware to combine several SOA technologies using different implementations of the component SOA. This combination is achieved by extending the component Plan Repository with a

component Plan Broker that federates the local Plan Repository with the components Service Discovery used to generate plans describing discovered remote services. In particular, the component Service Discovery encloses the service discovery protocols integrated in the middleware to advertise any newly discovered services to the Plan Repository [7]. Plans for these remote services are generated based on contracts negotiated by the component SLA Negotiation when discovered so that they are available when the Adaptation Manager initiates an adaptation at a later time. Plans are automatically discarded and removed from the Plan Repository when remote services disappear or for some reason become unavailable to the middleware.

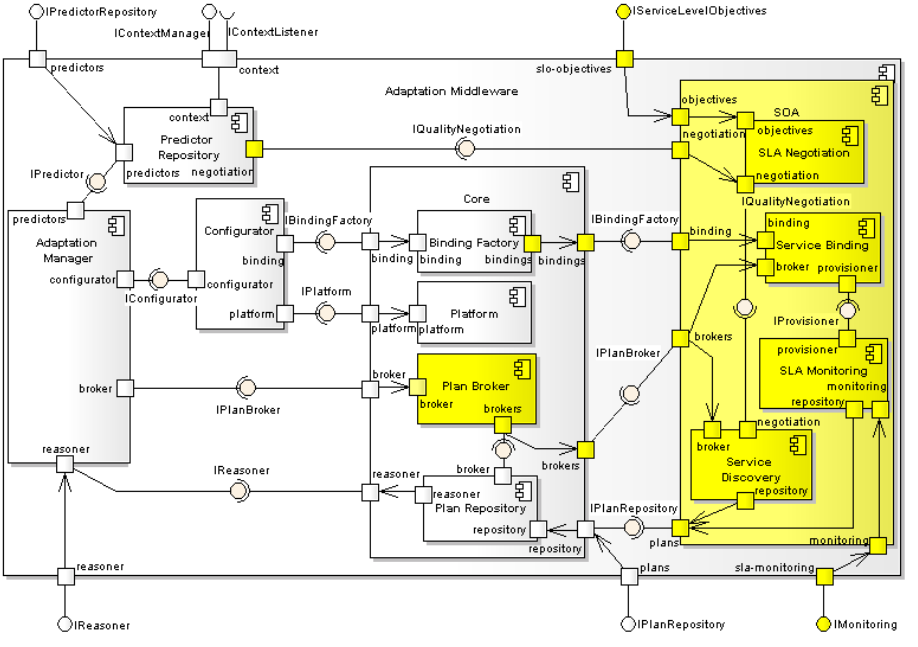

Fig. 2. The Architecture of the MUSIC planning-based adaptation middleware

SLA contracts can be either static or allow for some dynamic negotiation [2]. One example is a service level described by the service provider as QoS properties that are available at either static or negotiable cost. Furthermore, a service may offer a predefined set of service levels. When such a service is detected by the component Service Discovery, it generates a new service plan enclosing structural and behavioral metadata related to the service (*e.g.*, interface type description and contracts). Then, for each service level associated to this service, the component Service Discovery publishes an extended version of the service plan into the Plan Repository to reflect the alternative service levels available. This service level plan inherits from the metadata of the service plan and completes it with the additional QoS properties described by the service level (*e.g.*, service accuracy and cost).

4.2 Plan Reasoning

The component Adaptation Manager is then able to take into account each set of service levels when applying the reasoning heuristics. For planning to be efficient, service negotiation is a time critical factor that should be resolved as soon as possible. In our middleware, the negotiation is generally *static*, meaning that the negotiation is performed during service discovery for static QoS properties (*e.g.*, service cost) described by the service levels. The resulting static QoS property values are included into the service plan so that the property predictors can automatically report them at a later time.

However, in presence of a flexible service level [1], the negotiation becomes *dynamic*, meaning that the SLA contract is negotiated during the planning process. Dynamic negotiation is particularly useful when the Adaptation Manager needs to reason about up-to-date QoS properties (*e.g.,* current service accuracy). In this case the property predictors, when invoked by the reasoning heuristics, delegate to the component SLA Negotiation the negotiation of the requested property. The negotiation protocol is driven by *Service Level Objectives* (SLOs) [8] configured via the interface IServiceLevelObjectives. These objectives act as pre-defined criteria for negotiating a SLA contracts. As an example, the agent's company can define an SLO to minimize the response time of a service without exceeding its daily phone budget. Furthermore, the property predictor integrates a cache mechanism to reduce the latency of the negotiation protocol. This means that if two flexible service levels evaluated by the planning framework refer to the same QoS property of the associated service, then the negotiation protocol will be executed only once and the result of this negotiation will be considered valid for all the service levels associated with this service during the planning process.

Finally, the utility of application configurations using these service level plans will also get compared to application configurations based on plans associated with components, which can be locally deployed on the device, as well as plans representing the instance of the component or service already used by the application. The reasoning heuristics will therefore provide a uniform ranking of alternative application configurations, and the Adaptation Manager will select and deploy the configuration predicting the highest utility.

4.3 Plan Deployment and Configuration

As mentioned in Section 3.1, the component Configurator generally iterates over the plans composing the new application configuration to reconfigure the application. The support of remote services implies that it can now face three different situations. If the plan refers to an instance of a component, which is already used by the current application, the Configurator reuses this instance in the new configuration. If the plan refers to a component for which no instance is currently available, the Configurator uses the component Platform, to create and deploy a new instance of the component. Finally, if the plan refers to a remote service available in the environment, the Configurator uses the component Service Binding to generate a specific component

that will act as a *service proxy*[1]. A service proxy is a local representative of the remote service accessed by the application. In particular, it implements the service type described by the application components and encapsulates the communication protocol used to access the remote service. During this binding phase, the SLA contract associated with the selected plan is provisioned and enforced by the involved parties. This includes the reservation of computing resources by the provider and the deployment of SLA monitoring facilities [8]. This means that the SLA contract associated to the selected service is transferred to the component SLA Monitoring.

The service proxy implements also a *disconnection detection algorithm* due to the ubiquitous aspect of the application. This disconnection support is inspired from the principles of Ambient programming [9]. When loosing the connection to a remote service, the proxy stores the incoming service requests in a queue and returns a *non-blocking future object* to the application. The future object includes a block of code that is triggered when the service connection is resolved to process the result of the request. If the connection is lost for a long period, the service proxy breaks the SLA contract via the component SLA Negotiation. This notification triggers an adaptation of the application, and transfers the request queue to the new component (or service proxy) that will be planned and deployed.

Finally, the service proxy is also responsible for *monitoring the dynamic QoS properties associated to the SLA contract* agreed with the service [10]. To do so, the service proxy collects metrics at runtime (*e.g.*, the service response time) and reports the observed values to the component SLA Monitoring. This component is responsible for breaking the SLA contract if the observed value violates the value agreed. An example of violation of this contract can be a response time observed by the service proxy above the threshold agreed in the SLA. In practice, the component SLA Monitoring removes the associated service level plan from the Plan Repository to trigger a new adaptation of the application.

5 Case Study

As a preliminary validation of our approach, in this section we present a case study based on the scenario described in Section 2.

The architecture of the CRM application is introduced in Fig. 3. It basically supports two alternative compositions. Both contain a component GUI that presents a graphical user interface on the smartphone and a component Main that embeds the application logic and binds the different functionalities together. Main interacts with the CRM service to retrieve calendar and customer information, and with a Route Planning service to find the shortest route to reach a meeting location as well as the estimated travel time. It also uses a Navigation service to provide navigation aid to the user and a messaging service to alert affected customers about delays. In composition *a)* the navigation service is provided by a component Navigation deployed on the smartphone, which displays a map, the recommended route, and the current location on the smartphone display using the GUI. This component Navigation depends on a Map service and a Location service provided by a third party service provider. In

[1] Service proxy component bytecode is generated at runtime using the ASM bytecode manipulation framework (cf. http://asm.objectweb.org). The implementation details are out of the scope of this paper.

composition *b)* the Navigation service is provided by a third party service provider, supposed to use the provider's display to show the same information to the user. The QoS properties and service types relevant for the case study are specified in Table 1 and 2. Property predictors for the application, specified as functions of the properties of the services the application depends on, are associated with the compositions in Fig. 3.

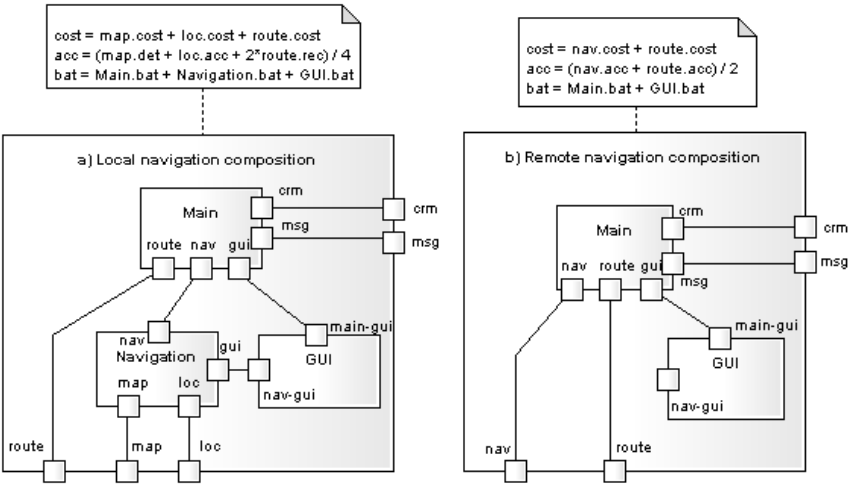

Fig. 3. The architecture model of the CRM client application

Table 1. The relevant QoS properties defined in the CRM client application

Property Name	Description	Value range
cost	Cost of using the service	0-?
acc	Accuracy, for example of a location	1-10
det	Level of detail of a map	1-10
rec	Recency of traffic info	1-10
bat	Battery units consumed by a component	1-100

Table 2. The service types defined in the CRM client application

Service Name	Description	Requested properties
loc	Locates the device geographically	cost, acc
map	Provides a map of a limited area	cost, det
route	Establish the fastest route between two locations and estimates the travel time	cost, rec
nav	Provides navigation aid	cost, acc

The landscape of remote services and how it evolves through the scenario is described in Fig. 4. The services are described by QoS properties that, together with the resources needed for communicating with them, determine the adaptation of the application.

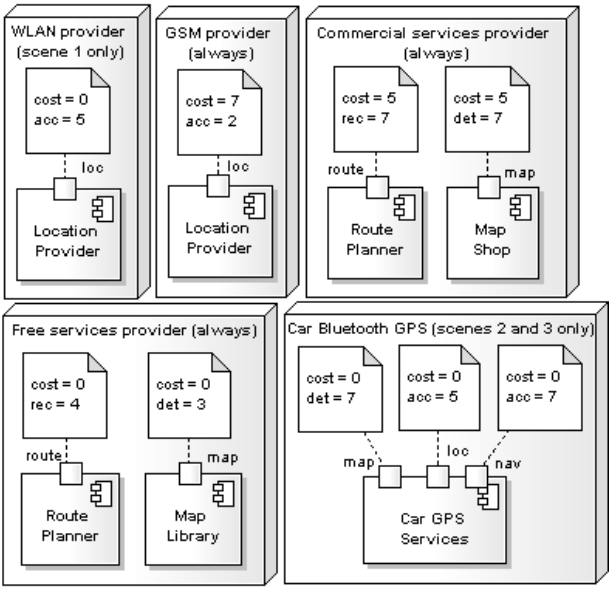

Fig. 4. The service landscape for the CRM client application

We did a simulation of the adaptation reasoning on this use case. In the simulation we also took into account estimation of the power consumption of the different alternatives, based on predictors for usage of memory, CPU, and network bandwidth. For the sake of simplicity[2], we used a simplified utility function assuming that the user prefers low cost (*i.e.*, to minimise cost), high accuracy (*i.e.*, to maximise acc), and needs to save battery (*i.e.*, to minimise bat). Thus, we define the function evaluating the utility of a CRM application configuration as the weighted sum of the normalised QoS properties (using the function *norm(…)*):

$$utility = \frac{user_{cost}}{norm(cost)} + user_{acc} \times norm(acc) + \frac{user_{bat}}{norm(bat)}$$

Table 3 summarizes the computed utility of the best configurations in different situations during the scenario.

Table 3. The CRM alternative configurations with their associated normalized utilities

Configuration				Utility	
Composition	loc	map	route	Scene 1&2	Scene 3
Local navigation	WLAN	Commercial	Commercial	**0,42**	0,42
Local navigation	WLAN	Free	Free	0,40	0,35
Remote navigation	Bluetooth	Free	Free	**0,66**	0,54
Remote navigation	Bluetooth	Free	Commercial	0,58	**0,58**

[2] In this scenario, we do not demonstrate the direct impact of memory, CPU, or network bandwidth variations on the computed utility values.

During the initial scene of the scenario, when the agent is in a meeting at a customer site, we observe that among the configurations available in this situation, composition *i)* using a WLAN Location service, binding to the commercial route planning service predicts the highest utility and is therefore chosen. In scene 2, when entering the car, the services provided by the car navigation system are discovered by the middleware component Bluetooth Service Discovery, which publishes the associated service plans (including *position accuracy* and *battery usage* as static QoS properties / *response time* as a dynamic QoS property) into the Plan Repository. After the agent has driven out of reach of the customer WLAN, the WLAN Location service plan is discarded from the Plan Repository and thus triggers the planning process. The configuration based on composition *ii)*, the Navigation service of the car GPS and the Free Route Planning service predicts the highest utility among the possible configurations. The adaptation middleware therefore reconfigures the CRM application to this configuration by generating Navigation and Route service proxies. In scene 3, the accuracy of the Free Route Planning service drops from 4 to 1. The service proxy observes this and notifies the component SLA Negotiation, which triggers a re-planning. The drop in accuracy of the Free Route Planning service causes the utility of the running configuration to fall below the predicted utility of the corresponding configuration with the Commercial Route Planning service. Therefore the Adaptation Manager selects this configuration and asks the Configurator to perform the reconfiguration of the service binding of the application.

6 Related Work

Adaptive Service Grids (ASG) is an open initiative that enables dynamic composition and binding of services, which is used for provisioning adaptive services [11]. In particular, ASG proposes a sophisticated and adaptive delivery service composed of three sub-cycles: *Planning*, *binding*, and *enactment*. The entry point of this delivery lifecycle is a *semantic service request*, which consists of a description of what will be achieved and not which concrete service has to be executed. Compared to our planning-based middleware, ASG focuses only on the planning per request of service workflows with regards to the properties defined in the semantic service request. Thus, ASG does not support a uniform planning of both components and services as our planning-based framework for ubiquitous applications does. However, we think that our planning-based middleware can be extended to integrate ASG adaptive services and seamlessly support dynamic enactment of service workflows that can provide the services required by a ubiquitous application.

Menasce and Dubey [12] propose an approach to QoS brokering in SOA. Consumers request services from a QoS broker, which selects a service provider that maximizes the consumer's utility function subject to its cost constraint. Utility functions express the usefulness of a system as a function of several attributes, such as response time, throughput, and availability. The approach assumes that service providers register with the broker by providing service demands for each of the

resources used by the services provided as well as cost functions for each of the services. The QoS broker uses analytic queuing models to predict the QoS values of the various services that could be selected under varying workload conditions. This approach is of interest from both the viewpoint of a consumer and a provider. While the client is relieved from performing service discovery and negotiation, the provider is given support for QoS management. The approach, however, requires the client device to be able to access the broker, but this might not be possible in mobile environments. It also assumes that the consumer is able to determine the expected service properties. Our approach differs in that it considers the offered properties as alternatives to determine the best application configuration.

CARISMA is a mobile computing peer-to-peer middleware exploiting the principle of reflection to support the construction of context-aware adaptive applications [13]. Services and adaptation policies are installed and uninstalled on the fly. CARISMA can automatically trigger the adaptation of the deployed applications by detecting execution context changes. CARISMA uses utility functions to select application profiles, which is used to select the appropriate action for a particular context event. If there are conflicting application profiles, then CARISMA proceeds to an auction-like procedure to resolve (both local and distributed) conflicts. Contrary to MUSIC, CARISMA does not deal with the discovery of remote services that can trigger application reconfigurations. However, the auction-like procedure used by CARISMA could be integrated in the MUSIC middleware as a particular implementation of the component SLA Negotiation.

ReMMoC is a dynamic middleware that supports interoperability between mobile clients and ubiquitous services [14]. During run-time, the ReMMoC service discovery component reconfigures itself and the remote service binding to match the protocols of the discovered ubiquitous services. Like MUSIC, ReMMoC uses architecture specifications for both the initial configuration and reconfigurations. However, ReMMoC does not support anything like service planning or discovery of service implementation alternatives, but applies rule-based policies that are limited to a fixed set of static component compositions.

7 Conclusion and Perspectives

In this paper we have introduced the design of a QoS-driven generic planning framework for self-adaptive mobile applications, which seamlessly supports and mixes component-based and service-based configurations. In particular, we have shown that the framework is able to adapt to changes in a landscape of ubiquitous remote services that may dynamically come and go, and where the offered service quality may vary. The framework exploits these changes to maximize the overall utility of applications. To that aim, the paper has shown how the planning middleware evaluates discovered remote services as alternative configurations for the functionalities required by a mobile application. The planning framework deals directly with SLA protocols supported by the services to negotiate the best quality of service for the user.

As a preliminary validation of our approach, the paper also explained how the planning framework handles a use case scenario in which a CRM application of sales agents exploits ubiquitous services, such as a location service, map service and traffic information service to improve the utility of the CRM application whenever such services are available.

In our future work, the presented planning framework will be realized as part of the MUSIC project. The framework will be validated using real world pilot applications of the industrial partners of the MUSIC project (http://www.ist-music.eu).

Acknowledgements

Thanks to partners of the MUSIC project and reviewers of the SC symposium for valuable comments. This work was partly funded by the European Commission through the project MUSIC (EU IST 035166). The scenario was inspired by a demonstrator application developed in the OSIRIS project (ITEA 04040 – http://www.itea-osiris.org) to evaluate the OSIRIS service platform.

References

1. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice-Hall, Englewood Cliffs (2006)
2. Dan, A., Ludwig, H., Pacifici, G.: Web service differentiation with service level agreements. IBM White Paper. pages 24 (May 2003)
3. Alia, M., Eide, V.S.W., Paspallis, N., Eliassen, F., Hallsteinsen, S., Papadopoulos, G.A.: A Utility-based Adaptivity Model for Mobile Applications. In: Proceedings of the 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW), pp. 556–563. IEEE, Niagara Falls, Ontario, Canada (2007)
4. Floch, J., Hallsteinsen, S., Stav, E., Eliassen, E., Lund, K., Gjørven, E.: Using architecture models for runtime adaptability. IEEE Software 23(2), 62–70 (2006)
5. Brataas, G., Hallsteinsen, S., Rouvoy, R., Eliassen, F.: Scalability of Decision Models for Dynamic Product Lines. In: International SPLC Workshop on Dynamic Software Product Line (DSPL). Kyoto, Japan, pages 10 (September 2007)
6. Erl, T.: SOA: Principles of Service Design. Prentice-Hall, Englewood Cliffs (2007)
7. Flores-Cortés, C.A., Blair, G.S., Grace, P.: An Adaptive Middleware to Overcome Service Discovery Heterogeneity in Mobile Ad Hoc Environments. IEEE Distributed Systems Online 8(7), 1 (2007)
8. Keller, A., Ludwig, H.: The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. Journal of Network and Systems Management 11(1), 53–81 (2003)
9. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-Oriented Programming. In: Companion of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (2005)
10. Morgan, G., Parkin, S., Molina-Jimenez, C., Skene, J.: Monitoring Middleware for Service Level Agreements in Heterogeneous Environments. In: Proceedings of the 5th IFIP conference on e-Commerce, e-Business, and e-Government (I3E), Poznan, Poland, October 26-28, 2005, vol. 189, pp. 79–93 (2005)

11. Fahringer, T., et al.: Adaptive Service Grids, White Paper. Deliverable (March 2007), http://asg-platform.org
12. Menasce, D., Dubey, V.: Utility-based QoS Brokering in Service Oriented Architectures. In: Proceedings of the International Conference on Web Services (ICWS), Salt Lake City, Utah (July 9-13, 2007)
13. Capra, L., Emmerich, W., Mascolo, C.: CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. IEEE Transactions on Software Engineering 29(10), 929–945 (2003)
14. Grace, P., Blair, G., Samuel, S.: ReMMoC: A Reflective Middleware to Support Mobile Client Interoperability. In: Meersman, R., Tari, Z., Schmidt, D.C. (eds.) CoopIS 2003, DOA 2003, and ODBASE 2003. LNCS, vol. 2888, pp. 1170–1187. Springer, Heidelberg (2003)