Context-Aware Adaptive Services: The PLASTIC Approach*

Marco Autili, Paolo Di Benedetto, and Paola Inverardi

Dipartimento di Informatica - Università degli Studi di L'Aquila, Italy
{marco.autili,paolo.dibenedetto,inverard}@di.univaq.it

Abstract. The near future envisions a pervasive heterogeneous computing infrastructure that makes it possible for mobile users to run software services on a variety of devices, from networks of devices to stand-alone wireless resource-constrained ones. To ensure that users meet their non-functional requirements by experiencing the best Quality of Service according to their needs and specific contexts of use, services need to be context-aware and adaptable. The development and the execution of such services is a big challenge and it is far to be solved. In this paper we present our experience in this direction by describing our approach to context-aware adaptive services within the IST PLASTIC project. The approach makes use of CHAMELEON, a formal framework for adaptive Java applications.

1 Introduction

Pervasive computing is an emerging paradigm that is rapidly changing the ways we use technologies to perform everyday tasks. The wide spread of small computing devices and the introduction of new communication infrastructures make it possible for mobile users to run software services on a variety of devices from networks of devices to stand-alone wireless resource-constrained ones. B3G networks [30] have gained importance as an effective way to realize pervasive computing by offering broad connectivity through various network technologies pursuing the convergence of wireless telecommunication and IP networks (e.g., UMTS, WiFi and Bluetooth).

Ubiquitous networking empowered by B3G networks makes it possible for mobile users to access networked software services across heterogeneous infrastructures through resource-constrained devices characterized by their *heterogeneity* and *limitedness*. Software applications running over this kind of infrastructure must cope with resource scarcity and with the inherent faulty and heterogeneous nature of this environment [9]. Indeed to ensure that users meet their non-functional requirements by experiencing the best Quality of Service (QoS) according to their needs and specific contexts of use, services need to be context-aware and adaptable. The development and the execution of such services is a big challenge for the

* This work is part of the IST PLASTIC project and has been funded by the European Commission, FP6 contract number 026955, http://www.ist-plastic.org/.

M. Chechik and M. Wirsing (Eds.): FASE 2009, LNCS 5503, pp. 124–139, 2009.

research community and it far to be solved. This paper describes our experience in this direction and, by extending our preliminary work in [17], presents our approach to context-aware adaptive services within the IST PLASTIC project [21]. The approach makes use of CHAMELEON, a formal framework for adaptive Java applications [7]. The goal of the PLASTIC project is the rapid and easy development, deployment and execution of adaptable services for B3G networks [30]. PLASTIC builds on Web Services (WS) and component-based technologies, and introduces the notion of *requested Service Level Specification* (SLS) and *offered SLS* to deal with the non-functional dimensions - i.e., QoS - that will be used to establish the *Service Level Agreement* (SLA) between the service consumer and the service provider. Services are implemented as adaptable components and are deployed on heterogeneous resource-constrained mobile devices. The new contribution of this paper is to describe the two types of adaptation supported by a PLASTIC service, namely adaptation driven by the (requested) SLS and adaptation with respect to the characteristic of the execution context. We name these two types of adaptation *SLS-based adaptation* and *context-aware adaptation*, respectively. Service adaptability is achieved via a development paradigm based on SLS and resource-aware programming supported by CHAMELEON that takes into account the characteristics of the hosting environment like resource availability, network conditions, and the SLSs.

The paper is organized as follows: Section 2 defines the two dimensions of the PLASTIC adaptation and Section 3 briefly describes the PLASTIC development environment. Section 4 introduces the CHAMELEON framework and describes how it has been used for implementing the PLASTIC Service-oriented Interaction Pattern. Section 5 describes the actual implementation of CHAMELEON by showing how the approach has been applied to the PLASTIC e-Health Remote Diagnosis case study. Related work is discussed in Section 6. Concluding remarks and future directions are given in Section 7.

2 PLASTIC Adaptation(s)

In this section we define the two dimensions of adaptation supported by a PLASTIC service.

The first kind of adaptation we consider is *SLS-based adaptation*. For this adaptation the context of interest is represented by the preferences expressed by the user in the *requested SLS* and by the provider in the *offered SLS*. The SLS represents the non-functional characteristics of the service. It is coupled with the service interface and it is used to establish the *SLA* between a user requesting the service and a provider of the service. The SLA defines the conditions on the QoS accepted by both the service consumer and the service provider. Adaptation in PLASTIC is used by the service provider to exhibit a service with different SLS. The exposed service is actually a generic one that at "matching time" is adapted with respect to the requested SLS and the available execution context.

The second type of adaptation is *context-aware adaptation*. For this adaptation the context of interest is represented by the provider, network and consumer contexts which represent the environment in which a service is provisioned and

consumed. PLASTIC applications are deployed over heterogeneous, resource-constrained devices, thus the provider and consumer contexts are their respective device resource characteristics - e.g., screen resolution, CPU frequency, memory size, available radio interfaces, networks in reach. The B3G network context identifies the characteristics of the (multiple) network(s) between consumer and provider such as number and type of networks, number of active users, number of available services, security, transmission protocol, access policy, etc. The network context impacts on the network QoS in terms of bitrate, transfer delay, packet-loss, network coverage, price and energy consumption for using a given network. Adaptation in this case is practiced by the service programmer who can produce a generic service that can be customized with respect to the actual resource characteristics of the execution context so that its execution can be correctly supported.

In PLASTIC adaptation is restricted at discovery time, that is at the moment in which the service execution context and the user QoS preferences (requested SLS) are known, and a SLA can be put in place. The advantage of this approach to adaptation is that it is cost effective since it is a compromise between static and fully dynamic adaptation. The limit is that unpredictable changes of contexts might invalidate the SLA and thus make the service unusable. However in highly heterogeneous and autonomous infrastructures like B3G, QoS attributes cannot be (a priori) guaranteed. Thus SLA violations can occur and must be monitored and detected. Indeed, in [17] we present a first attempt to tackle this problem by monitoring service execution to detect possible SLA violations. Upon violation either (*i*) the service can be adapted to the new context so that it can continue respecting the agreed SLA or (*ii*) a re-negotiation of the SLA can happen which can in turn drive a new adaptation.

3 PLASTIC Development Environment

In this section we briefly describe the PLASTIC development environment from the perspective of a service developer. PLASTIC provides a set of tools[1] that are all based on the *PLASTIC Service Conceptual Model*[2] and support the service life cycle, from design to implementation to validation to execution. The conceptual model formalizes all the concepts needed for developing B3G service-oriented applications. The overall approach is model driven, starting from the conceptual model till the execution service model used to monitor the service.

With reference to Figure 1, the PLASTIC conceptual model has been concretely implemented as a *UML2 profile* and, by means of the PLASTIC development environment tools, the functional behavior of the service and its non-functional characteristics can be modeled. Then, non functional analysis and development activities are iteratively performed [10]. The analysis aims at computing QoS indices of the service at different levels of detail, from early design

[1] Available at http://gforge.inria.fr/projects/plastic-dvp/

[2] The *Formal description of the PLASTIC conceptual model and of its relationship with the PLASTIC platform toolset* is available at http://www.ist-plastic.org/

to implementation to publication, to support designers and programmers in the development of services that satisfy the specified QoSs, i.e., SLSs. In PLASTIC, among QoS measures, we only consider performance and reliability. The principal performance indices are *utilization* and *throughput* of (logical and physical) resources, as well as *response time* for a given task. The considered reliability measures are, instead, *probability of failure on demand* and *mean time to failure*. Discrete set of values - e.g., high, medium, low - are used to identify ranges.

The analysis and validation activities rely on artifacts produced from the PLASTIC service model through different model transformations. For instance, the service model editor [8] and the SLA editor, that are part of the PLASTIC platform toolset, are integrated through a model-to-code transformation. Once the service model has been specified, a model-to-code transformation can be performed in order to translate the parts of the service model that are needed for specifying the agreement (e.g., involved parties, other services, operations, etc.) into a HUTN file (i.e., a human-usable textual notation for SLA) which the SLA editor is capable to import. The SLS attached to the published service and the SLA are formally specified by using the language SLAng [16].

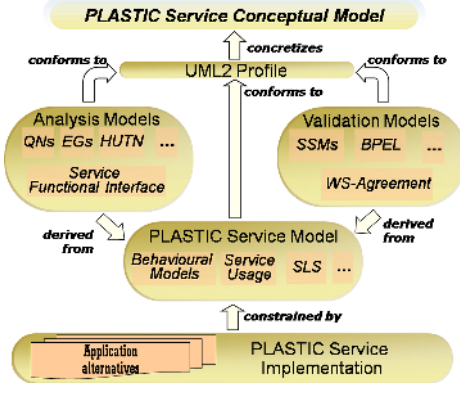

Fig. 1. Development Environment

After the service has been implemented, the PLASTIC validation framework enables the off-line, prior to the service publication, and on-line, after the service publication, validation of the services with respect to functional - through test models such as Symbolic State Machines (SSMs) or based on BPEL processes - and non-functional properties [12]. This means that through validation it is possible to assess whether the service exhibits the given SLS. On-line validation concerns the "checking" activities that are performed after service deployment such as SLA monitoring [23].

For the purposes of this work, hereafter we will concentrate on how PLASTIC services are implemented and how the two types of adaptation, namely, *SLS-based adaptation* and *context-aware adaptation*, are supported.

4 PLASTIC Services Deployment and Access

PLASTIC services are implemented by using the CHAMELEON *Programming Model* (presented in Section 5) that, extending the Java language, permits developers to implement services in terms of *generic code*. Such a generic code, opportunely preprocessed, generates a set of different *application alternatives*, i.e., different standard Java components that represent different ways of

implementing a provider/consumer application. Therefore, an adaptable software service might be implemented as and consumed by different application alternatives (i.e., different adaptations). Each alternative is characterized by (*i*) the resources it demands to be correctly executed (i.e., *Resource Demand*) and (*ii*) the so called *Code-embedded SLSs*. The latter are QoS indices retrieved by the non-functional analysis. They are specified by the developers at generic code level through annotations attached to methods and are then automatically "injected" into the application alternatives by CHAMELEON. As it will be clear in Section 5, code-embedded SLSs contribute to determine the final SLSs offered by the different alternatives.

In the remainder of this section we describe how adaptive PLASTIC services are published, discovered and accessed (Section 4.1), and how both provider- and consumer-side application alternatives (stored in the *Applications Registry*) can be over-the-air delivered and deployed on devices (Section 4.2).

4.1 The PLASTIC Service-Oriented Interaction Pattern

The PLASTIC Service-oriented Interaction Pattern for provision and consumption of adaptive services (Figure 2) involves the following steps.

The service provider publishes into the *PLASTIC Registry* the service description in terms of both functional specifications and associated offered SLSs (1). Specifically, a provider can publish a service with different SLSs, each one associated to a different provider-side application alternative that represents a way of adapting the service. The service consumer queries the PLASTIC registry for a service functionality, additionally specifying the requested SLS (2). The PLASTIC registry searches for service descriptions that satisfy the consumer request. If suitable service descriptions are present in the service registry, the service consumer can choose one of them on the base of their offered SLSs. After the service consumer accepts an offered SLS, the registry returns the actual reference to the provider-side application alternative that implements the (adapted) service with the accepted SLS (3.a). Thus, the SLA can be established and the service consumption can take place (4). If no suitable published service is able to directly and fully satisfy the requested SLS, negotiation is necessary (3.b). The negotiation phase starts by offering a set of alternative SLSs. The consumer can accept one of the proposed SLSs, or perform an "adjusted" request by reiterating the process till an SLA is possibly reached. In Figure 2 the box *SLA* labeling the provider and the consumer represents the agreement reached by both of them.

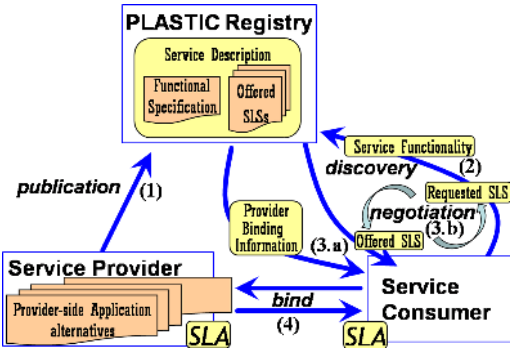

Fig. 2. PLASTIC Interaction Pattern

4.2 Over-the-Air Application Alternatives Delivery and Deployment

With reference to Figure 3, we call *PLASTIC-enabled devices* the devices deploying and running the CHAMELEON *Client* component and the *PLASTIC B3G Middleware* [13] that together are able to retrieve contextual information. A PLASTIC-enabled device provides a declarative description of the execution context in terms of the resources it supplies (i.e., *Resource Supply*) and a description of the impact that computational elements (i.e., code instructions) have on the resources (i.e., *Resource Consumption Profile*). PLASTIC-enabled devices can host both service consumers and service providers.

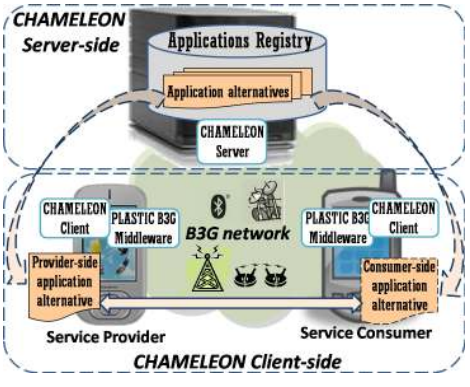

Fig. 3. CHAMELEON Client-Server

Indeed, the CHAMELEON *Client* component can interact with the CHAMELEON *Server* component in order to dynamically download (from the CHAMELEON *Applications Registry*), deploy and run (*i*) provider-side application alternatives, e.g., a .war file for a web application to be exposed as service and, if needed, (*ii*) ad-hoc consumer-side application alternatives, e.g., a .jar and a .jad files for a midlet to be used for consuming a service. Note that, the CHAMELEON client is a lightweight component that effortlessly runs on limited devices; the CHAMELEON server runs on a back-end server which does not suffer resource limitations.

By referring to Figure 4, (*i*) a provider that wants to offer a service *S* can connect directly to the CHAMELEON application registry and search for an application among a set of already implemented application alternatives to be exposed as service *S*. The choice is based on the functional description of *S* and the code-embedded SLSs restricting to those alternatives that are compatible (i.e., will run safely) with respect to the execution context (i.e., resource supply and resource consumption profile). The chosen alternative will be automatically delivered and deployed via the Over-The-Air (OTA) provisioning

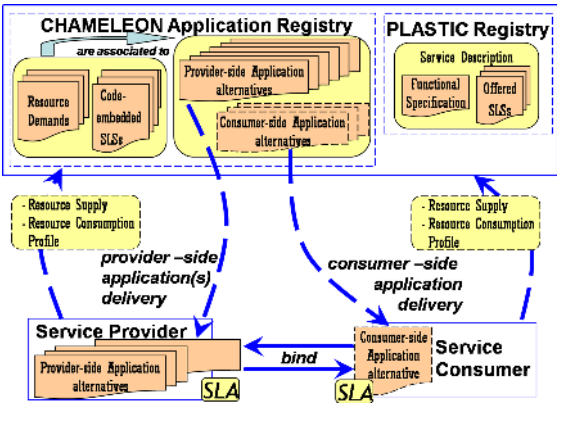

Fig. 4. Application Alternatives Delivery

technique [2] on the provider device. Note that, the process can be used for delivering and deploying more than one alternative $A_1, \ldots, A_n$ for S. Then, the functional description of S will be published into the PLASTIC registry along with the final offered SLSs defined on the base of the code-embedded SLSs, associated to $A_1, \ldots, A_n$, possibly refined by the provider through the SLA editor (see Section 3).

(*ii*) Differently, if to consume the service S an ad-hoc client application needs to be deployed on the consumer device, the final offered SLSs published by the provider will be defined on the base of the code-embedded SLSs associated to all the chosen provider application alternatives $A_1, \ldots, A_n$ combined with the code-embedded SLSs associated to all the consumer application alternatives of S. In this case, upon the consumer request, the PLASTIC registry relies on the CHAMELEON application registry to search for a set of compatible consumer-side application alternatives $C_1, \ldots, C_m$ that are able to safely run on the consumer device and properly interact with the service S (i.e., $A_1, \ldots, A_n$). The delivered consumer application alternative will depend on the SLS chosen among the only offered SLSs related to $C_1, \ldots, C_m$.

5 CHAMELEON-Based PLASTIC Services Implementation

In this section we present the CHAMELEON framework and show how it supports the implementation of PLASTIC adaptive services and their provision and consumption. For more detailed presentations of the formalisms and definitions underlying the framework please refer to [6,7] (and references therein). The CHAMELEON framework has been implemented [7] on the Java platform and it exploits XML-based technologies for data exchange. The framework will be presented by means of the PLASTIC e-Health Remote Diagnosis case study.

The e-Health service allows to establish a link between patients and assistants providing support for video conferences, medical agenda management, alarm generation and management, remote diagnosis (RD), etc. Both professionals and patients are considered to be nomadic and can move with their mobile devices e.g., move from outdoor to their office/home. Therefore, mobility becomes a key issue and services need to be adapted, both to the heterogeneous networks capabilities and to the terminals that could be used by professionals and patients.

For the purposes of this paper, we focus on the RD functionality. When an alarm is generated on the patient side due to some event like patient inactivity, dangerous vital parameters or help request, the e-Health system contacts one or more doctors to perform a diagnosis. On the doctor side the diagnosis process is supported by an RD consumer application that, connecting to an RD service provider installed on the patient side, allows the doctor to check the patient camera and monitor vital parameters, e.g., blood pressure, temperature, heart rate. The whole service is adapted according to both the patient (the provider) and doctor (the consumer) context.

► **Programming Model**. Referring to right-hand side of Figure 5, the *Development Environment* (DE) is based on a *Programming Model* that provides

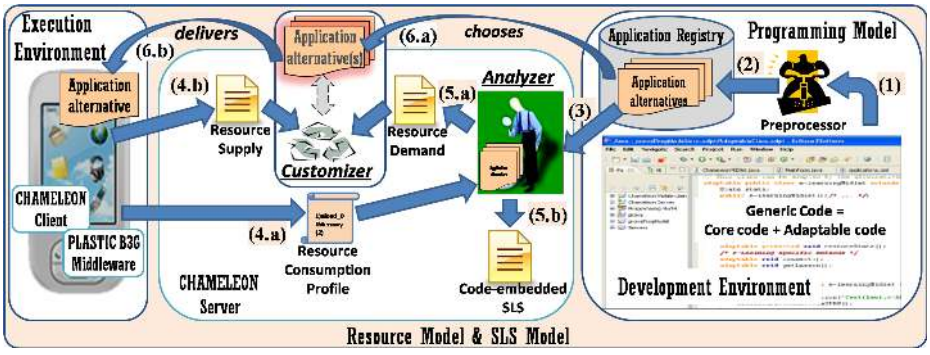

Fig. 5. CHAMELEON Framework

developers with a set of ad-hoc extensions to Java for easily specifying services code in a flexible and declarative way. As already mentioned, services code is a *generic* code that consists of two parts: the *core* and the *adaptable* code - see in Figure 5 the screen-shoot of our DE implemented as an Eclipse plugin [7]. The core code is the frozen portion of the application and represents its invariant semantics. The adaptable one represents the degree of variability that makes the code capable to adapt. The generic code is preprocessed by the CHAMELEON *Preprocessor* (1), also part of the DE, and a set of different standard Java application alternatives is automatically derived and stored into the *Application Registry* (2).

Figure 6 represents an excerpt of a generic code, as part of the RD consumer MIDlet, written by the developer according to the CHAMELEON programming model. The core code is a standard code and, hence, can be specified through standard Java classes; the adaptable code is an "extended" code and is specified through *Adaptable Class*es that declare one or more *Adaptable Method*s. Methods are the smallest building blocks that constitute the entry-points for a behavior that can be adapted. *Alternative Class*es define how one or more adaptable methods can actually be adapted. For instance, the adaptable class RemoteDiagnosis declares three adaptable methods (see the keyword **adaptable**): visualCheck, vitalParameters and connect. The implementation of these adaptable methods is defined by two alternative classes (see the keywords **alternative** and **adapts**): HighSupportRD and LowSupportRD. Such a generic code will be preprocessed by the CHAMELEON *Preprocessor* and the two standard Java application alternatives described in Table 1 will be derived by suitably combining the adaptable methods implementations specified by the various alternatives.

The programming model also allows for specifying additional information by using *Annotation*s. Annotations are specified at the generic code level and permit to specify resource demand (*Resource Annotation*), code-embedded SLS (*SLS Annotation*), upper bound on the number of loop iterations (*Loop Annotation*) and recursive method calls (*Call Annotation*).

```
adaptable public class RemoteDiagnosis extends MIDlet {
  ...
  adaptable void connect();
  adaptable void visualCheck();
  adaptable void vitalParameters();
  ...
}
/*****************************************************************************/
alternative class HighSupportRD adapts RemoteDiagnosis {
  private void connect() { ...
    Annotation.SLSAnnotation(''Throughput(high)'');
    Annotation.resourceAnnotation(''WiFi(true)'');
    QoSInfo.setBitrate(HIGH);
    PlasticMiddleware.selectNetwork(QosInfo);
  }
  private void visualCheck() {/*shows video streaming from patient's cameras*/}
  private void vitalParameters() {/*draws diagrams of vital parameters*/}
}
/*****************************************************************************/
alternative class LowSupportRD adapts RemoteDiagnosis {
  private void connect() {  ...
    Annotation.SLSAnnotation(''Throughput(low)'');
    QoSInfo.setBitrate(LOW);
    PlasticMiddleware.selectNetwork(QosInfo);
  }
  private void visualCheck() {/* shows patient's camera images */ }
  private void vitalParameters() {/*shows textual data of vital parameters*/}
}
```

Fig. 6. An Adaptable MIDlet

Table 1. Remote Diagnosis Consumer Application Alternatives

Alternative	Features	Resource Demand	Code-embedded SLS
Cons_1	Shows patient's camera images and textual data of vital parameters	{Energy(200)}	{Throughput(low)}
Cons_2	Shows video streaming and images from patient's cameras and draws diagrams of vital parameters	{WiFi(true), Energy(400)}	{Throughput(high)}

Annotations can be specified by calls to "do nothing" static methods of the Annotation class in Figure 7. For instance, in Figure 6 the method calls Annotation.resourceAnnotation("WiFi(true)") and Annotation.SLSAnnotation("Throughput(high)") are used to specify that the HighSupportRD alternative class demands for a WiFi radio-interface on the consumer device and provides a high quality remote diagnosis support. Note that, a high throughput is related to the usage of the resource WiFi. These annotations will contribute to determine the resource demand and the code-embedded SLSs, respectively, of the derived alternatives. Indeed, the whole framework is based on the *Resource* and *SLS Model*s (see Figure 5) that, in particular, allow for specifying conforming resource and SLS annotations, respectively.

```
public class Annotation {
   public static void resourceAnnotation(String ann){};
   public static void SLSAnnotation(String ann){};
   public static void loopAnnotation(int n){};
   public static void callAnnotation(int n){};
}
```

Fig. 7. Annotation Class

► **Resource and SLS Models**. The resource model is a formal model that allows the characterization of the resources needed to consume/provide a service and it is at the base of context-aware adaptation. The SLS model is a model that

permits developers to attach non-functional information at generic code level through code-embedded SLSs and is used for SLS-based adaptation purposes (see Section 4).

A resource is modeled as a typed identifier that can be associated to natural, boolean or enumerated values. Natural values are used for consumable resources whose availability varies during execution (e.g., energy, heap space). Boolean values define non-consumable resources that can be present or not (e.g., function libraries, network radio interfaces) and enumerated values define non-consumable resources that provide a restricted set of admissible values (e.g. screen resolution, network type). Figure 8 shows an example of some resource and SLS definitions for the RD case study. Both the *Resource Demand* and the *Resource Supply* are specified in terms of *resource sets* that couple resources to their values in the form $\{res_1(val_1), ..., res_n(val_n)\}$. Table 1 also reports the resource demand and the code-embedded SLS calculated by the analyzer (see below). For example, the resource demand of the Cons_2 application alternative in Table 1, specifies that, to run safely, the alternative will require a WiFi network readio-interface (`WiFi(true)`) and a battery state-of-charge of the target consumer device at least of 400 energy units (`Energy(400)`).

Resource Definition
defineRES Energy **as** Natural
defineRES Bluetooth **as** Boolean

SLS Definition
defineSLS Throughput **as** {low, medium, high}
defineSLS Mobility
as{low, medium, high}

Fig. 8. Resource and SLS Definitions

The *SLS Model* bases itself around the same formalisms as the resource model and it is used for specifying SLSs. For example, the code-embedded SLS of the Cons_2 alternative of Table 1, specifies that the alternative offers a high throughput (`Throughput(high)`).

► **Chameleon Server**.Still referring to Figure 5, the *Analyzer* (running on the CHAMELEON server) is an interpreter that, abstracting a standard JVM, is able to analyze the application alternatives (3) and derive their resource consumption (5.a) and the code-embedded SLSs (5.b). The analyzer is parametric with respect to the characteristics of the execution environment as described through the *resource consumptions profile* sent by the device (4.a). We remind that the profile provides a characterization of the target execution environment, in terms of the impact that Java bytecode instructions have on the resources. Note that this impact depends on the execution environment since the same bytecode instruction may require different resources in different execution environments.

More precisely, these profiles associate resources consumption to particular patterns of bytecode instructions specified as *regular expressions*. Since the bytecode is a verbose language[3], this allows to define the resource consumption associated to both basic instructions (e.g., `ipush`, `iload`, etc.) and complex ones, e.g., method calls. Figure 9 represents an example of a resource consumption profile. For instance, the last row states that a call to the `getLocalDevice()`

[3] This is particularly true for method invocations where the method is uniquely identified by a fully qualified *id* (base class identifier + name + formal parameters).

1) aload_0 → {Memory(2)} **2)** .* → {Memory(1), Energy(1)}
3) invokestatic LocalDevice.getLocalDevice() → {Bluetooth(true), Energy(20)}

Fig. 9. A Resource Consumption Profile

static method of the `LocalDevice` class within the `javax.bluetooth` library requires the presence of Bluetooth on the device (`Bluetooth (true)`), and it causes a consumption of the resource Energy equal to 20 cost units.

The analyzer performs a worst-case analysis of the program by statically inspecting the Java bytecode of the different application alternatives. More specifically, the analyzer scans each possible execution path of the application alternative bytecode by reconstructing (through the DAVA decompiler[4]) and traversing the bytecode abstract syntax tree (BAST). Within a path, each encountered instruction is matched against the resource consumption profile and, by combining the demand of each instruction, the overall resource demand of the path is derived. The final resource demand (5.a) of the application alternative will be the one of the most demanding execution path. At the same time the encountered SLS annotations contribute to determine the code-embedded SLS (5.b). The analyzer is based on an operational semantic that has been formalized using a transition system. It is out of the scope of this paper to go into details of our analysis technique, and we refer to [6] for further details.

$$\frac{\begin{array}{c} IsLeaf(n) \quad Label(n) = \texttt{instr} \\ \texttt{instr}\ !Like(\texttt{"invoke*"}) \\ !IsAnnotation(n) \\ r = b(\texttt{instr}) \end{array}}{\langle e, b, M, n\rangle \rightarrow_{ARA} \{\langle r, \phi\rangle\}}$$

Fig. 10. Fall-Back Leaf Rule

As an example, in Figure 10 we show a very simple rule that is applied when the transition system (reaching a BAST leaf node n) encounters a basic bytecode instruction `instr` that is neither a method invocation (i.e., $!Like(\texttt{"invoke*"})$) nor an annotation (i.e., $!IsAnnotation(n)$). The rule simply uses the function b that matches `instr` against the resource consumption profile in order to obtain its resource demand r. The result is a set of pairs $\langle r, s\rangle$ where s contains the encountered code-embedded SLSs (empty in this case since `instr` is not an annotation).

Still referring to Figure 5, the resource demands (5.a) of the application alternatives together with the resource supply sent by the device (4.b) are used by the *Customizer* that is able to choose (6.a) and propose a set of "best" suited application alternatives, and deliver (6.b) consumer- and/or provider-side standard Java applications that (through the delivery mechanism described in Section 4.2) can be automatically deployed in the target devices for execution. The customizer bases on the notion of *compatibility* that is used to decide if an application alternative can run safely on the requesting device, i.e., if for every resource demanded by the alternative a "sufficient amount" is supplied by the execution environment. For instance, a resource supply {`WiFi(true), Energy (300)`} would be compatible only with the resource demand of adaptation Cons_1 in Table 1.

[4] Available at http://www.sable.mcgill.ca/dava/

Table 2. Remote Diagnosis Provider Application Alternatives

Alternative	Features	Resource Demand	Code-embedded SLS
Prov_1	Transmits images from patient camera, stores and makes available manually inserted vital parameters values	{GPRS(true), Memory(128), ...}	{Throughput(low), Mobility(high)}
Prov_2	Streams video from patient camera, collects and makes available data automatically retrieved by measurement instruments, and provides all functionalities of alternative Prov_1	{Memory(512), PressureMeter(true), HRM(true), WiFi(true)...}	{Throughput(high), Mobility(medium)}

Now, let us assume that the two application alternatives in Table 2 have been derived for the provider (i.e., the patient): Prov_1 provides a high patient mobility[5] (`Mobility(high)`) but a low throughput (`Throughput(low)`), Prov_2 provides a limited patient mobility but a high throughput.

Since to consume the RD service an ad-hoc client application needs to be deployed on the consumer device (see Section 4.2), the code-embedded SLSs associated to provider alternatives that will be stored in the CHAMELEON application registry will be those in Table 3. They results from the combination of the code-embedded SLSs associated to the provider alternatives in Table 2 with the ones associated to the consumer alternatives in Table 1. Note that merging `Throughput(low)` with `Throughput(high)` produces `Throughput(low)`. Still referring to Section 4.2, let us now assume that a patient (a provider), deploying both the alternatives Prov_1 and Prov_2 of Table 2, has published the RD Service in the PLASTIC registry along with the offered SLSs associated to them in Table 3. Upon a doctor (a consumer) request, the PLASTIC registry relies on the CHAMELEON server to obtain a set of applications compatible with the doctor device. If the doctor device is compatible only with the consumer-side RD alternatives Cons_1 of Table 1, the doctor will be allowed to choose among only the offered SLSs related to Cons_1 (i.e., the combined SLSs of Cons_1-Prov_1 and Cons_1-Prov_2 in Table 3).

Table 3. Combined Code-embedded SLSs

	Cons_1	Cons_2
Prov_1	{Throughput(low), Mobility(high)}	{Throughput(low), Mobility(high)}
Prov_2	{Throughput(low), Mobility(medium)}	{Throughput(high), Mobility(medium)}

6 Related Work

Our resource model and analyzer can be related to other approaches to resource-oriented analysis. The MRG Project [5] proposes a framework (based on proof-carrying code) for giving correct guarantees that programs are free from run-time violations of resource bounds. The MRG approach is based on a resource-counting semantics that takes into account, through a resource component, the number of executed instructions and the maximum size of the stack frame. The same line of research is continued in the Mobius project [11]. For example, in [3]

[5] Mobility describes the size of network coverage considered by the PLASTIC B3G middleware.

the authors propose static analysis framework for the cost analysis of sequential Java bytecode that adds cost relations for defining the cost of a program as a function of its input data size.

In [25] the authors present a framework that, at both deployment- and runtime, is able to estimate the energy consumption of a distributed Java-based system. In particular, at the component level, they integrate an energy cost model and a communication cost model that allow for estimating the overall energy cost of each component. This information can then be used by software engineers in order to make decisions when adapting an application.

Tivoli [19] provides a resource modeler tool that enables the specification of resources and allows for the automatic monitoring of them by instrumenting the environment in which programs are executed.

The worst-case execution-time (WCET) problem has been deeply studied in the literature. In [28], the authors give an exhaustive survey of methods and tools for estimating (under precise assumptions) the WCET of hard real-time systems. Even though not strictly related to our work, this work provides an in-depth insight into the static and dynamic program analyses techniques used in this research area. In particular, the work in [4] also addresses the WCET problem and proposes a parametric timing analysis that instead of computing a single numeric value for WCET, as done by numeric timing analyses, derives symbolic formulas for representing the WCET. By accounting for parameters of the program, processor behaviour, and by deriving parametric loop bounds, the proposed analysis allows for deriving precise WCETs.

All these approaches use a resource model and aim at giving an absolute (over-)estimation of resources' consumption. Our approach instead, does not aim at establishing a punctual estimation rather aims at supporting a reasoning mechanism for selecting alternatives. Its correctness thus is restricted to consistently reflect the ordering among resource consumptions of a set of alternatives that will run in the same execution context.

By exploring all the possible computation paths and by mapping JVM bytecode to a transition system, the CHAMELEON analyzer uses the same exhaustive technique of other existent tools such as Java Pathfinder (JPF) [26]. JPF checks for property violations (deadlocks or unhandled exceptions) traversing all possible execution paths. Differently from JPF, our analyzer gets rid of variable values and abstracts the JVM with respect to resource consumption. In this abstraction we consider bytecode instructions behavior only by taking into account their effects on resources.

For the sake of space, we cannot address all the recent related works in the wide domain of software services. In the following, we provide only some major references. In [20] an interesting approach is presented that considers separation of concerns between application logic and adaptation logic. The approach makes use of Java annotations to express metadata needed to enable dynamic adaptation. Stemming from the same separation of concerns idea, in [24] in the scope of the MUSIC project [1], the authors propose the design of a middleware- and architectural-based approach to support the dynamic

adaptation and reconfiguration of the components and service composition structure. They use a planning-based middleware that, based on metadata included in the available plans, enables the selection of the right alternative architectural plan for the current context. Similarly to us this approach is based on requested and offered QoS, and supports SLA negotiation. Differently from us, they do not consider adaptability to the device execution context basing on a resource oriented analysis that is parametric with respect to resource consumption profiles.

Current (Web-)service development technologies, e.g., [14,15,22,27,29] (just to cite some), address only the functional design of complex services, that is they do not take into account the extra-functional aspects (e.g., QoS requirements) and the context-awareness. Our process borrows concepts from these well assessed technologies and builds on them in order to make QoS issues, context-awareness and adaptiveness emerge in service development.

7 Discussion and Future Work

In this paper we have described how the Chameleon framework is used to realize a form of service adaptation in the IST PLASTIC project. Chameleon allows for the development and deployment of adaptable applications (consuming and providing services) targeted to mobile resource-constrained devices in the heterogeneous B3G network. We have proposed a Service-oriented Interaction Pattern for adaptable service provision and consumption, and have described how it is based on Chameleon to support the two-fold PLASTIC adaptation from both service provider side and consumer side.

Right now, PLASTIC adaptation happens at discovery time, thus the deployed application is customized (i.e., it is tailored) with respect to the context at binding time but, at run time, it is frozen with respect to evolution. If context changes at run time the service needs to dynamically adapt to continue respecting the reached SLA and dependability. As first attempt to tackle this problem, in [17] we propose a mechanism that, exploiting ad-hoc methods for *saving* and *restoring* the (current) application state, enables services' evolution against monitored SLA violations. Evolution is achieved by dynamically un-deploying the no longer apt alternative and subsequently (re-)deploying a new alternative that is able to preserve the agreed SLA. If no alternative is able to continue respecting the agreed SLA, a re-negotiation of the SLA can happen which can in turn drive a new adaptation.

Moreover, assuming that upon service request the user knows (at least a stochastic distribution of) the mobility pattern [18] they will follow during service usage, this permits to identify the successive finite contexts they can traverse during service usage. Following this approach, an enhanced version of Chameleon would be able to generate code that is a compromise between self-contained (i.e., embedding adaptation logic) and tailored adaptable code. The code would merge the adaptation alternatives that, associated to the specified mobility pattern, are necessary to preserve the *offered SLS*. The code will also embed some dynamic *adaptation logic* which is able to recognize context changes, seamlessly

"switching" among the embedded adaptation alternatives. In this way, a seamless evolution will be performed among the embedded alternatives associated to the mobility pattern, while the un-/re-deployment evolution will be performed when moving out of the mobility pattern's context.

We are currently performing an empirical analysis of the framework in order to evaluate its correctness and its applicability to more complex real scenarios. Moreover, we are investigating how ontology based specifications might be used to establish a common vocabulary and relationships among resource/SLS types. This would allow to relate resource/SLS types and to predicate about a common set of related types. For instance, a demand of `Energy` could be related to a supply of `Battery`.

References

1. MUSIC Project, `http://www.ist-music.eu/`
2. Over-The-Air (OTA), `http://developers.sun.com/mobility/midp/articles/ota/`
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost analysis of java bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
4. Altmeyer, S., Hümbert, C., Lisper, B., Wilhelm, R.: Parametric Timing Analyis for Complex Architectures. In: Proc. of the 14th IEEE RTCSA, pp. 367–376. IEEE Computer Society Press, Los Alamitos (2008)
5. Aspinall, D., MacKenzie, K.: Mobile resource guarantees and policies. In: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (2006)
6. Autili, M., Di Benedetto, P., Inverardi, P.: Resource oriented static analysis of Java programs. Technical Report univaq-1243 (2008), `http://www.di.univaq.it/chameleon/output/download.php?fileID=1243`
7. Autili, M., Di Benedetto, P., Inverardi, P., Mancinelli, F.: Chameleon project - SEA group, `http://di.univaq.it/chameleon/`
8. Autili, M., Berardinelli, L., Cortellessa, V., Di Marco, A., Di Ruscio, D., Inverardi, P., Tivoli, M.: A development process for self-adapting service oriented applications. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 442–448. Springer, Heidelberg (2007)
9. Autili, M., Caporuscio, M., Issarny, V.: A reference model for service oriented middleware. Technical Report inria-00326479, INRIA Paris-Rocquencourt (2008)
10. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE TSE 30(5), 295–310 (2004)
11. Barthe, G.: Mobius, securing the next generation of java-based global computers. In: ERCIM News (2005)
12. Bertolino, A., De Angelis, G., Di Marco, A., Inverardi, P., Sabetta, A., Tivoli, M.: A Framework for Analyzing and Testing the Performance of Software Services. In: Proc. of the 3rd ISoLA. CCIS, vol. 17, Springer, Heidelberg (2008)
13. Caporuscio, M., Raverdy, P.-G., Moungla, H., Issarny, V.: ubiSOAP: A service oriented middleware for seamless networking. In: Proc. of 6th ICSOC (2008)
14. Eclipse.org. Eclipse Web Standard Tools, `http://www.eclipse.org/webtools`
15. IBM. BPEL4WS, Business Process Execution Language for Web Services (2003)
16. Skene, J., Lamanna, D., Emmerich, W.: Precise service level agreements. In: Proc. of the 26th ICSE, pp. 179–188, Edinburgh, UK (May 2004)

17. Autili, M., Di Benedetto, P., Inverardi, P., Tamburri, D.A.: Towards self-evolving context-aware services. In: Proc. of CAMPUS (DisCoTec), vol. 11 (2008)
18. Di Marco, A., Mascolo, C.: Performance analysis and prediction of physically mobile systems. In: Proc. of WOSP, NY, USA, pp. 129–132 (2007)
19. Moeller, M., Callahan, B., Gucer, V., Hollis, J., Weber, S.: Introducing Tivoli Distributed Monitoring Workbench 4.1. IBM Redbooks (2002)
20. Paspallis, N., Papadopoulos, G.A.: An approach for developing adaptive, mobile applications with separation of concerns. In: COMPSAC (2006)
21. PLASTIC project, `http://www.ist-plastic.org`
22. A-MUSE Project. Methodological Framework for Freeband Services Development (2004), `https://doc.telin.nl/dscgi/ds.py/Get/File-47390/`
23. Raimondi, F., Skene, J., Emmerich, W.: Efficient Online Monitoring of Web-Service SLAs. In: Proc. of the 16th ACM SIGSOFT/FSE (November 2008)
24. Rouvoy, R., Eliassen, F., Floch, J., Hallsteinsen, S.O., Stav, E.: Composing components and services using a planning-based adaptation middleware. In: Pautasso, C., Tanter, É. (eds.) SC 2008. LNCS, vol. 4954, pp. 52–67. Springer, Heidelberg (2008)
25. Seo, C., Malek, S., Medvidovic, N.: An energy consumption framework for distributed java-based systems. In: ASE (2007)
26. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. ASE journal 10(2) (2003)
27. W3C. Web Service Definition Language, `http://www.w3.org/tr/wsdl`
28. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. Trans. on Embedded Computing Sys. 7(3), 1–53 (2008)
29. Yun, H., Kim, Y., Kim, E., Park, J.: Web Services Development Process. In: Proc. of Parallel and Distributed Computing and Systems (PDCS) (2005)
30. Zahariadis, T., Doshi, B.: Applications and services for the B3G/4G era. Wireless Comm. 11(5) (2004)