Run time models in adaptive service infrastructure

Marco Autili, Paola Inverardi, Massimo Tivoli

Run time models in adaptive service infrastructure

Marco Autili, Paola Inverardi and Massimo Tivoli

Abstract. Software in the near ubiquitous future will need to cope with variability, as software systems get deployed on an increasingly large diversity of computing platforms and operates in different execution environments. Heterogeneity of the underlying communication and computing infrastructure, mobility inducing changes to the execution environments and therefore changes to the availability of resources and continuously evolving requirements require software systems to be adaptable according to the context changes. Software systems should also be reliable and meet the user's requirements and needs. Moreover, due to its pervasiveness, software systems must be dependable. Supporting the validation of these self-adaptive systems to ensure dependability requires a complete rethinking of the software life cycle. The traditional division among static analysis and dynamic analysis is blurred by the need to validate dynamic systems adaptation. Models play a key role in the validation of dependable systems, dynamic adaptation calls for the use of such models at run time. In this paper we describe the approach we have undertaken in recent projects to address the challenge of assessing dependability for adaptive software systems.

1. Introduction

The near future envisions a pervasive heterogeneous computing infrastructure that makes it possible to provide and access software services on a variety of devices, from end-users with different needs and expectations. 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

This work is a revised and extended version of [28]. It has been partially supported by the IST project PLASTIC No 026955, the FET project CONNECT No 231167, and the Italian Prin D-ASAP. We acknowledge all the members of the consortium for PLASTIC, CONNECT, and D-ASAP, and of the SEALab at University of L'Aquila for joint efforts on all the research efforts reported in this paper.

context-aware, adaptable, and dependable. The development and the execution of such services is a big challenge and it is far to be solved.

Software services and systems in the near ubiquitous future (Softure) cannot rely on the classical desktop-centric assumption that the system execution environment is known a priori at design time and, hence, the execution environment of a Softure cannot be statically anticipated. Softure will need to cope with variability, as software systems get deployed on an increasingly large diversity of computing platforms and operates in different execution environments. Heterogeneity of the underlying communication and computing infrastructure, mobility inducing changes to the execution environments and therefore changes to the availability of resources and continuously evolving requirements require software systems to be adaptable according to the context changes.

Context awareness and adaptation have become two key aspects to be considered while developing and running future software. While providing/consuming services, Softure need to be aware of and adaptive to their context, i.e., the combination of user-centric data (e.g., requested QoS, information of interest for users according to their current circumstance) and resource/computer-centric data (e.g., resources and conditions of devices and network). At the same time, Softure should be dependable and meet the users performance requirements and needs. Moreover, due to its pervasiveness and in order to make adaptation effective and successful, adaptation must be considered in conjunction with dependability, i.e., no matter what adaptation is performed, the system must continue to guarantee a certain degree of QoS.

Supporting the development and execution of Softure systems raises numerous challenges that involve languages, methods and tools for the systems through design, analysis, and validation in order to ensure dependability of the self-adaptive systems that are targeted.

In this paper we describe three completely different instantiations of a development process model for Softure systems. We report our experience in assessing dependability for adaptive software systems within the context of three different projects. Although completely different, it is worthwhile considering the three scenarios since each of them focus on different dependability requirements. However, due to their different nature and aims, neither a common example cannot be used to discuss them nor it is meaningful for them to give an integrated view.

The first one, that we will describe in detail, concerns the IST PLASTIC [36] project that can be considered as a specific instance of Softure as context-aware software for next generation networking environments. Another scenario concerns the FET CONNECT [21] project that aims at enabling continuous composition of networked systems to respond to the evolution of functionalities provided to and required from the networked environment. Since the project is still in a preliminary stage, we cannot provide a detailed description of it. In the last scenario we describe how to manage the performance of the Siena publish/subscribe middleware [16, 19] by using the Performance Management Framework (PFM). PFM allows the management of the system performance at run time based on monitoring and model-based performance evaluation [17]. We will keep also the description of the PFM scenario short since it concerns a particular activity of a development process model for Softure, i.e., dynamic reconfiguration driven by the run-time exploitation of performance models.

Our thesis is that Softure requires to rethink the whole software development process since it never stabilizes, but it is permanently under maintenance. Software evolution, which is traditionally practiced as an off-line activity, must often be accommodated at run time for Softure. This requires to reconcile the static view with the dynamic view by breaking the traditional division among development phases by moving some activities from design time to deployment and run time hence asking for new and more efficient verification and validation techniques. Dependability is achieved with a comprehensive life cycle approach from requirements to operation, to maintenance by analyzing *models*, testing code, monitor, and repair execution. Many software models are involved, from requirements to specification, to code. In order to support dependability of adaptable applications new modeling notations are required. These should permit to express and deal with characteristics that are crucial for a Softure, i.e., QoS, resource-awareness, evolution, reconfiguration, variability, and uncertainty. At the same time they should allow for validation techniques affordable at run time. Their cost must be sustainable under the execution environment resource constraints, e.g. time and computational resources. In order to easily and consistently integrate the modeling layer with the analysis and implementation ones, model transformation and evolution techniques should be exploited.

The paper is structured as follows. In the following section we discuss the Softure characteristics in order to identify the two key challenges, i.e., *adaptability* and *dependability*, and to highlight the crucial role that models play, at run time, for adaptive software. This discussion will bring us to consider the Softure issues in a software process perspective. In Section 3, we propose a new software process for Softure, where models play a key role in dynamic adaptation and dependability validation. In Section 4, we show the application of the proposed process to three different scenarios we borrowed from three projects we participate(d) to. In Section 5 we conclude by summarizing the thesis originating from the discussion carried on throughout the paper.

2. Setting the context

Softure is supposed to execute in an ubiquitous, heterogeneous infrastructure under mobility constraints. This means that the software must be able to carry on operations while changing different execution environments or *contexts*. Execution contexts offer a variability of resources that can affect the software operation. *Context awareness* refers to the ability of an application to *sense* the context in which it is executing and therefore it is the base to consider (self-)adaptive applications, i.e., software systems that have the ability to change their *behavior* in response to external changes.

It is worthwhile stressing that although a change of context is measured in quantitative terms, an application can only be adapted by changing its behavior, i.e., its functional/qualitative specification. For instance, (Physical) Mobility allows a user to move out of his proper context, traveling across different contexts and, to our purposes, the difference among contexts is determined in terms of available resources like connectivity, energy, software, etc. However other dimensions of contexts can exist relevant to the user, system and physical domains, which are the main context domains identified in the literature [38, 9, 26, 7, 20, 40]. In the software development practice when building a system the context is determined and it is also part of the (non-functional) requirements (operational, social, organizational constraints). If context changes, requirements change therefore the system needs to change. In standard software, the pace at which context changes is slow and they are usually taken into account as evolutionary requirements. For Softure, context changes occur due to physical mobility while the system is in operation. This means that if the system needs to change this should happen dynamically. This notion leads to consider different ways to modify a system at run time that can happen in different forms namely *(self-)adaptiveness/dynamicity* and at different levels of granularity, from software architecture to line of code.

Softure needs also to be dependable. *Dependability* is an orthogonal issue that depends on QoS attributes, like performance and all other -*bilities*. Dependability impacts all the software life cycle. In general dependability is an attribute for software systems that operate in specific application domains. For Softure we consider dependability in its original meaning as defined in [27], that is *the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers ... Dependability includes such attributes as reliability, availability, safety, security.* Softure encompasses any kind of software system that can operate in the future ubiquitous infrastructure. The dependability requirement is therefore extended also to applications that traditionally have not this requirement. Dependability in this case represents the user requirement that states that the application must operate in the unknown world (i.e., out of a confined execution environment) with the same level of reliance it has when operating at home. At home means in the controlled execution environment where there is complete knowledge of the system behavior and the context is fixed. In the unknown world, the knowledge of the system is undermined by the absence of knowledge on contexts, thus the dependability requirement arises also for conventional applications. Traditionally dependability is achieved with a comprehensive approach all along the software life cycle from requirements to operation to maintenance by analyzing models, testing code, monitor and repair execution.

Therefore the overall challenge is to provide dependable assurance for highly adaptable applications. Since dependability is achieved throughout the life cycle many software artifacts are involved, from requirements specification to code. In the rest of this paper, as such artifacts, we will consider *models*, i.e., an idealized view of the system suitable for reasoning, developing, validating a real system. Models can be functional and non-functional and can represent different level of abstractions of the real system, from requirements to code. Our research bias is on Software Architecture, therefore we will often consider software architectural systems' models. An architectural model allows the description of the static and dynamic components of the system and explains how they interact. Software architectures support early analysis, verification and validation of software systems. Software architectures are the earliest comprehensive system model along the software lifecycle built from requirements specification. They are increasingly part of standardized software development processes because they represent a system abstraction in which design choices relevant to the correctness of the final system are taken. This is particularly evident for dependability requirements like security and reliability and quantitative ones like performance.

3. The Process View: the Role of Models at Run Time

In this section we cast the above discussed challenges in a development process view. The process view focuses on the set of activities that characterize the development and the operation of a software system. These activities are traditionally divided into activities related to the actual production of the software system and activities that are performed when the system can be executed and goes into operation. Specification, Design, Validation, and Evolution activities vary depending on the organization and the type of system being developed. Each Activity requires its Language, Methods and Tools and works on suitable artefacts of the system. For validation purposes each artefact can be coupled with a *model*. Models are an idealized view of the system suitable for reasoning, developing, validating a real system. To achieve dependability a large variety of models are used from behavioural to stochastic. These models represent the systems at very different levels of abstraction from requirements specification to code. The ever growing complexity of software has exacerbated the dichotomy development/static/compile time versus execution/dynamic/interpreter time concentrating as many analysis and validation activities as possible at development time.

Softure puts new requirements on this standard development process. The evolutionary nature of Softure makes infeasible a standard approach to validation since it would require before the system is in execution to predict the system behaviour with respect to virtually any possible change. Therefore, in the literature most approaches that try to deal with the validation of dynamic software system concentrate the changes to the structure by using graph and graph grammars formalisms or topological constraints [11, 23, 25, 30, 32, 39, 14, 22]. As far as changes to behaviour are concerned, only few approaches exist that make use either of behavioural equivalence checks or of the type system [2, 3, 4] or through code certification [12, 33]. If dependability has to be preserved through adaptation, whatever the change mechanism is, at the time the change occurs a validation check must be performed. This means that all the models necessary to carry on the validation step must be available at run time and that the actual validation time becomes now part of the execution time. In this direction, it is worth to mention the work in [24] where the authors present a model-driven approach (D-KLAPER) that, building on the KLAPER (Kernel LAnguage for PErformance and Reliability analysis) intermediate language, allows for capturing the core features for the performance and reliability analysis of a dynamically reconfigurable component-based system. Indeed, this approach goes into a similar direction as the projects later on described (specifically, PLASTIC and PFM) and, in particular, KLAPER has been also used in PLASTIC for performance analysis.

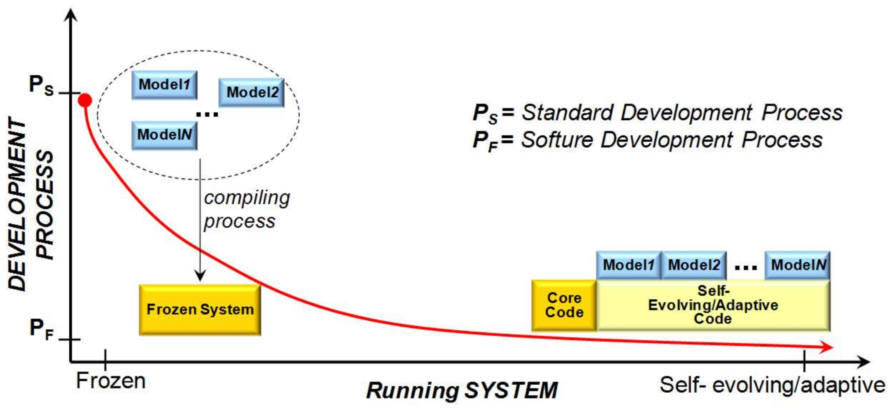


Figure 1. The Softure Development Process

The Softure development process therefore has to explicitly account for complex validation steps at run time when all the necessary information are available. Figure 1 represents the process development plane delimited on the left-hand side by the standard development process and on the right-hand side by the softure development one. The vertical dimension represents the static versus dynamic time with respect to all the activities (e.g., analysis and validation) involved in the development process. The horizontal axis represents the amount of adaptability of the system, that is its ability to cope with evolution still maintaining dependability. The standard development process carries out most of the development and validation activities before the system is running that is during development. The result is a running system that, at run time, is frozen with respect to evolution. Considering development processes that allow increasingly degrees of adaptability allows to move along the horizontal axis thus ideally tending to a development process that is entirely managed at run time. In the middle we can place development processes that allow larger and larger portions of the system to change at run time and that make use for validation purposes of artefacts that can be produced statically. In the following section we describe three instances of the Softure Development Process that have been proposed in the scope of the projects PLASTIC, CONNECT, and PFM respectively. Although completely different, and described at a different level of detail, these instances show how different dependability requirements can be considered and dynamically assessed. Note that PLASTIC and CONNECT, in particular, are two EU IP projects (STREP and FET respectively). Clearly, we cannot provide (and it is also out of the scope of this paper) the reader with neither a validation/comparison with respect to the state of the art nor a detailed treatment (e.g., assumptions and inherent limitations) of each single technique/method/tool developed within the specific project. The reader interested in these aspects can refer to [36] and [21] (and the on-line documentation therein). Moreover, we cannot describe in details all the models that, at different level and in different ways, are analysed and manipulated to contribute to evaluate/estimate (some of) the different dependability attributes. Rather, our aim is to report our attempts towards different instantiations of the Softure development process.

4. Instantiating the Process Model: 3 Scenarios

In this section we describe the approach we have undertaken in recent projects to address the challenge of assessing dependability for adaptive software systems. We consider three different scenarios. We will mainly concentrate on the first one that concerns the ended IST PLASTIC [36] project. It can be considered as a specific instance of Softure as context-aware software for next generation networking environments. Another scenario comes from the preliminary results of the recently started FET CONNECT [21] project. The CONNECT goal is to enable continuous composition of networked systems in order to respond to the evolution of functionalities provided to and required from the networked environment. In the last scenario, we shortly describe the PFM framework developed as part of a project internal to our group. In particular, we describe the application of PFM to the management of the performance of the Siena publish/subscribe middleware [16, 19]. PFM allows the management of the system performance at run time based on monitoring and model-based performance evaluation [17].

Before diving into the presentation of the three projects, it must be noted that the three experienced approaches will be described with different level of detail since they have a different size and maturity, and they are completely different in nature with respect to kind of models, their usage degree at run time, and their purposes. For instance, in PLASTIC (part of) the models are created at design time and (possibly) evolved at run time basing on the results of the analysis and monitoring activities. In the more open scenario of CONNECT, models are created at run time, through the learning activities, and exploited at run time for generating connectors on the fly. Moreover, some of the described results, at present, might be considered either a little bit outdated, for the ended projects

PLASTIC and PFM, or not enough mature, for the ongoing CONNECT project. For instance, the UML and the QoS profiles used within PLASTIC have both become obsolete and the more recent UML MARTE profile [35] can be found today. Again, it is out of the scope of this paper to embark on such a discussion.

4.1. The PLASTIC project

The main goal of the PLASTIC project is the rapid and easy development/deployment of context-aware adaptive services for the next generation pervasive networking environment, such as Beyond 3^{rd} Generation (B3G) networks [41]. B3G networks are 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. Indeed, to ensure that users meet their non-functional requirements by experiencing the best QoS according to their needs and specific contexts of use, services need to be context-aware, adaptable, and dependable.

This section describes our experience in this direction by describing the PLASTIC development process which integrates different engineering methods, tools and models for supporting a user-centric and comprehensive process (from design to deployment to validation). 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. To this end, the model proposes an elicitation of base abstractions that need to be accounted for developing applications for B3G networking infrastructures. Starting from the analysis of the characteristics of B3G networks, the relevant abstractions have been identified and refined according to the PLASTIC goals. Explicitly considering B3G networks, the *PLASTIC middleware* conveniently combines the heterogeneous wireless networks in reach, which for various reasons (e.g., cost effectiveness, distinct administration and infrastructureless ad hoc interaction) are not integrated at the network layer. With reference to Figure 2, where we show the service oriented interaction pattern, the PLASTIC middleware [18] provides enhanced support for the publication, discovery and access of services over B3G. Hence, the PLASTIC middleware supports services for being aware of the networking environment and for adapting to its changes. The middleware is able to capture the various networks and observe their status, and

[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/

abstract their properties in order to fully exploit the underlying network diversity without relying on any pre-established knowledge or infrastructure. To this end, the middleware exploits at run time a B3G *network context model* that introduces (in particular) the notion of `Network Infrastructural Path` between the service provider and consumer. Specifically, a network infrastructural path is a combination of (possibly different) wireless networks together with their network-level services. For instance, referring to Figure 2, a possible network infrastructural path is $WiFi_1 \Leftrightarrow UMTS_1 \Leftrightarrow WiFi_2$.

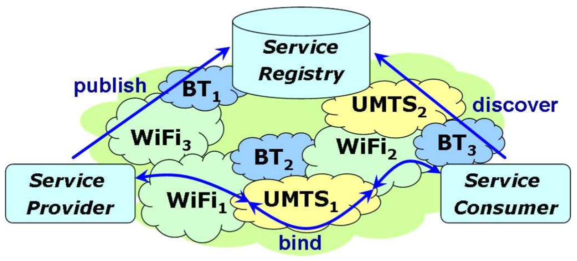


FIGURE 2. *Service Oriented Interaction Pattern over B3G*

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 3, 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 to implementation to publication, to support designers and programmers in the development of services that satisfy the specified QoSs, i.e., Service Level Specifications (SLSs). In PLASTIC, among QoS dimensions, we 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 obtained QoS indices are the base to guarantee the implementation of dependable services that satisfy the required/offered QoS. In particular, the SLS relies on information retrieved by the analysis of the quantitative models Queuing Networks (QNs - see Chapter 1) and Execution Graphs (EGs) derived from the service model. The kind of QNs we sue in PLASTIC is "product form QN".

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 Service Level Agreement (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 [29].

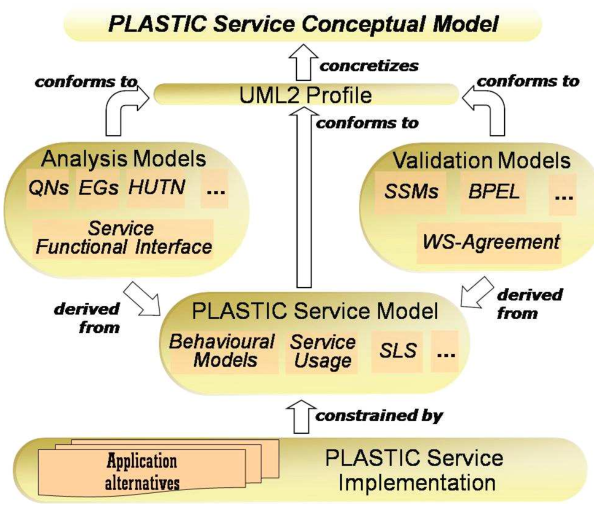


FIGURE 3. Development Environment

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 [13]. 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 [37].

PLASTIC builds on Web Services (WS) and component-based technologies, and (as one of the main novelties) 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 SLA between the service consumer and the service provider. Services are offered by adaptable components and are deployed on heterogeneous resource-constrained mobile devices. Adaptable components are implemented by using CHAMELEON, a formal framework for adaptive Java applications [5, 6, 31, 7]. By exploiting CHAMELEON the PLASTIC services support two types of adaptation, namely *SLS-based adaptation* driven by the (requested/offered) SLS and *context-aware adaptation* driven by the characteristic of the (provider, network and consumer) execution context.

PLASTIC services are implemented by using the Chameleon *Programming Model* [5, 6, 31, 7] 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. Upon service publication, code-embedded SLSs are then combined with the SLSs specified by the service provider hence contributing to determine the final SLSs offered by the different alternatives.

Figure 4 shows how adaptive PLASTIC services are published, discovered and accessed by describing the steps involved within the PLASTIC Service-oriented Interaction Pattern for the provision and consumption of adaptive services.

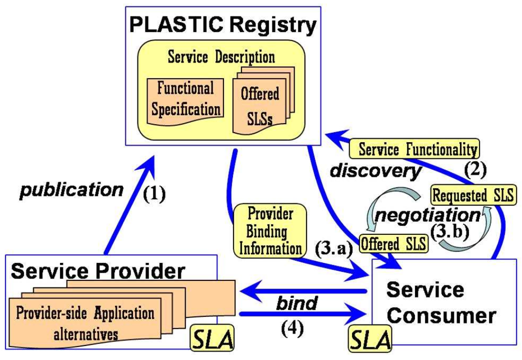


Figure 4. PLASTIC Interaction Pattern

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 4 the box *SLA* labeling the provider and the consumer represents the agreement reached by both of them.

Hereafter, we describe some of the (run time) models defined for an eHealth application that has been developed by using the PLASTIC methodologies and tools previously introduced. However, at the end of the section, we will briefly describe the CHAMELEON framework and show an excerpt of the CHAMELEON-based generic code, derived through model-to-code transformations, for the resource-aware adaptable components implementing the modeled `eHealth` service. A detailed description of the CHAMELEON framework can be found in [5, 6, 31, 7].

A PLASTIC eHealth application. The main purpose of the eHealth application is to attend persons, which live in countryside and widespread areas. In particular, it aims at distributing effectively existing assistant systems to mobile and remote patients and doctors, improving quality of service and system availability while giving those users even the benefits of disconnected mobile operations. The main capabilities of the application are depicted in Figure 5.

In case of an emergency, the patient sends an alarm that can be forwarded to available resources that are not included in customary eHealth service deployments, such as relatives to the patient, rural doctors and social workers (this is the `AlarmManagement` use case in the figure). This behaviour is permitted only once the patient has subscribed the service (`SeviceSubscription` use case). Moreover, while using the service, the family supervisor context can change because of structural modifications, mobility, new supervisor is assigned to the customer etc. Structural modifications can occur also to the patient side that is new devices can be deployed and, hence, new device drivers are required and suitable (with respect to the new devices) versions of the application service must be deployed (`ServiceAdaptation` use case). The rest of the section focuses on the `AlarmManagement` use case only even though it provides the reader with all the required information to understand the application of the PLASTIC development process and its tool support.

The architecture of the eHealth application is depicted in Figure 6. The patient is equipped with electronic devices and by means of a residential gateway it is in contact with specialists and relatives. In the `AlarmManagement` use case, the patient presses an alarm button and an alarm request is sent to the service

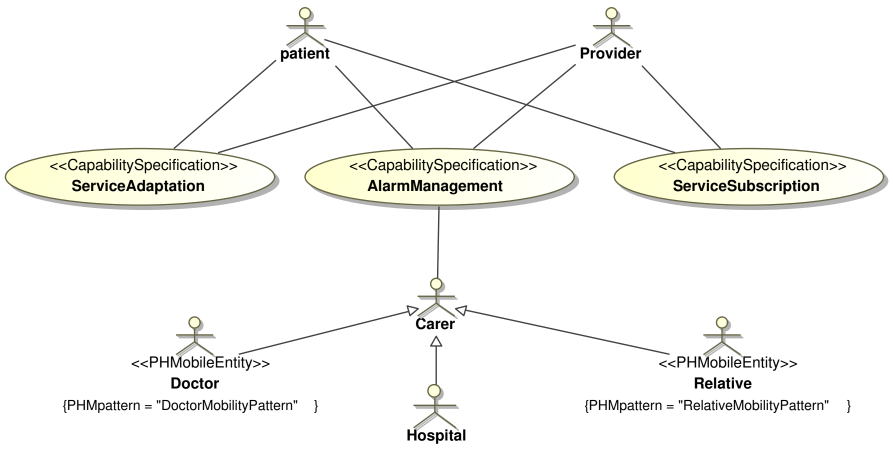

patient
Provider
<<CapabilitySpecification>>
ServiceAdaptation
<<CapabilitySpecification>>
AlarmManagement
<<CapabilitySpecification>>
ServiceSubscription
Carer
<<PHMobileEntity>>
Doctor
{PHMpattern = "DoctorMobilityPattern" }
Hospital
<<PHMobileEntity>>
Relative
{PHMpattern = "RelativeMobilityPattern" }

FIGURE 5. eHealth Use Cases

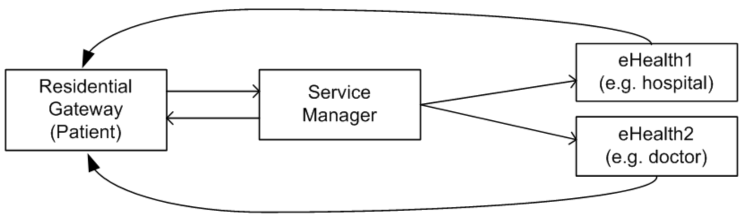


FIGURE 6. eHealth Architecture

manager through the residential gateway. The service manager attempts to notify `eHealth1` (in this case the hospital). If the request can be satisfied by the hospital an alarm notification is returned to the patient. Alternatively, the service manager tries the discovering of other registered eHealth services (e.g. a doctor or a relative) and sends them the alarm request taking into account a priority queue. An alarm notification is sent to the residential gateway by the reached eHealth service.

Following the PLASTIC development process, the specification of the eHealth application starts from the definition of the services that have to be aggregated or combined. For this purpose, the *service description diagram* in Figure 7 is produced. In this specification, `eHealthService` is the front-end to many small services which can be PLASTIC services that still have to be developed (in this case referred by `composed by` stereotyped dependencies) or already existing services which are simply used by the other ones and referred in the model by means of `uses` stereotyped dependencies. For instance, `eHealthService` uses the `Middleware` intended to be an always available service that exports the methods of the PLASTIC middleware which supports the discovery, provisioning, and access of context-aware services over devices providing heterogeneous B3G networks access.

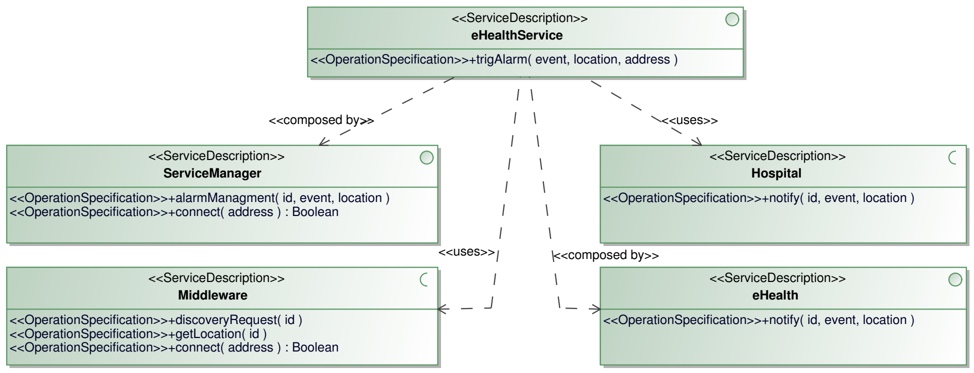
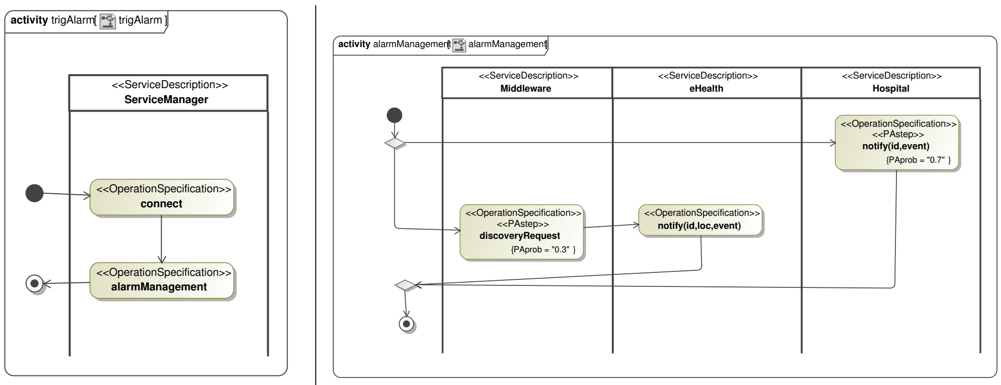

<<ServiceDescription>>
eHealthService
<<OperationSpecification>>+trigAlarm(event, location, address)
<<composed by>>
<<uses>>
<<ServiceDescription>>
ServiceManager
<<OperationSpecification>>+alarmManagment(id, event, location)
<<OperationSpecification>>+connect(address) : Boolean
<<ServiceDescription>>
Hospital
<<OperationSpecification>>+notify(id, event, location)
<<uses>>
<<composed by>>
<<ServiceDescription>>
Middleware
<<OperationSpecification>>+discoveryRequest(id)
<<OperationSpecification>>+getLocation(id)
<<OperationSpecification>>+connect(address) : Boolean
<<ServiceDescription>>
eHealth
<<OperationSpecification>>+notify(id, event, location)

FIGURE 7. Service Description Diagram

Once all services are defined, a number of *business process descriptions* have to be provided. In particular, for each capability of the larger services, a business process model has to be specified in order to describe the interactions among the involved smaller services. In Figure 8 the `trigAlarm` and `alarmManagement` capabilities of the `eHealthService` and `ServiceManager` services, respectively are modeled. In particular, when the patient presses the alarm button, the `trigAlarm` operation of the `eHealthService` service is invoked. This opens a connection with the `ServiceManager` and invokes the `alarmManagement` operation (see Figure 8.a). This operation is described in the right-hand side of the figure. According to the requirements summarized above, an alarm can be managed by the hospital (with a probability of `0.7` as specified by the non-functional annotation `PAprob` borrowed from the UML SPT Profile[3]). Alternatively, a relative or a doctor is notified (see the invoked `notify` operation of `eHealth` in the figure) after a discovery operation performed by exploiting the middleware facilities.

In general, a service can be implemented by one or more components and, in turn, a component can be used to implement one or more services. The PLASTIC approach proposes the use of the *service specification diagram* for defining the components implementing the "smaller" services. The diagram in Figure 9 specifies the sample components that implement the `eHealth` service which will be deployed on the devices of the relatives and the doctors. The services installed on the residential gateway, in the service manager, and in the hospital are not taken into account

[3]http://www.omg.org/cgi-bin/doc?formal/2005-01-02.



(a) trigAlarm (b) alarmManagement

Figure 8. Business Process Descriptions

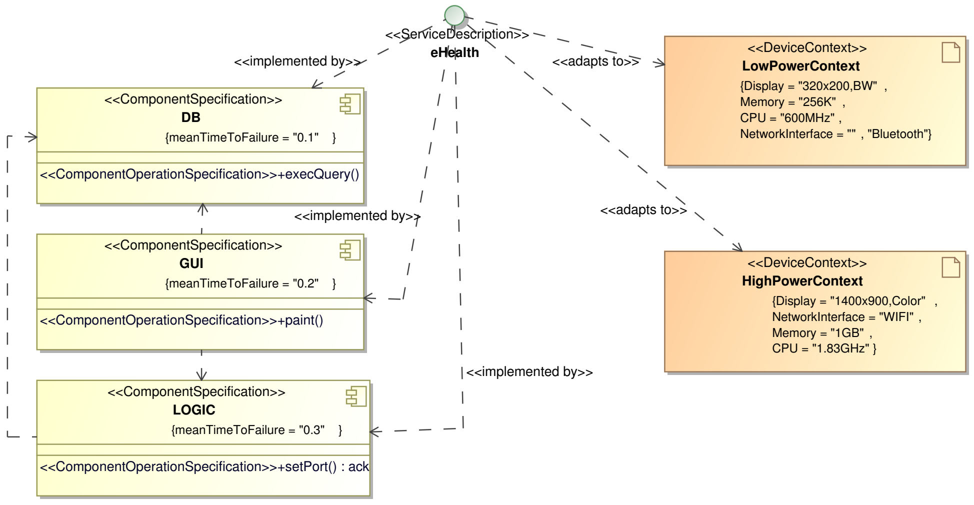

<<ServiceDescription>>
eHealth
<<implemented by>>
<<adapts to>>
<<DeviceContext>>
LowPowerContext
{Display = "320x200,BW" ,
Memory = "256K" ,
CPU = "600MHz" ,
NetworkInterface = "" , "Bluetooth"}
<<ComponentSpecification>>
DB
{meanTimeToFailure = "0.1" }
<<ComponentOperationSpecification>>+execQuery()
<<implemented by>>
<<adapts to>>
<<DeviceContext>>
HighPowerContext
{Display = "1400x900,Color" ,
NetworkInterface = "WIFI" ,
Memory = "1GB" ,
CPU = "1.83GHz" }
<<ComponentSpecification>>
GUI
{meanTimeToFailure = "0.2" }
<<ComponentOperationSpecification>>+paint()
<<implemented by>>
<<ComponentSpecification>>
LOGIC
{meanTimeToFailure = "0.3" }
<<ComponentOperationSpecification>>+setPort() : ack

Figure 9. Service Specification Diagram

here since they are not mobile and do not require context aware adaptations. According to Figure 9, `eHealth` is implemented by three sample components: `DB` is devoted to the storing of data managed by the application; `GUI` offers the operations for creating windows, buttons, dialogs, menus and everything required for user interactions; `LOGIC` is a component that implements the logic of `eHealth`. The diagram contains also the description of the contexts in which the service will be able to adapt. In particular, eHealth can be deployed on two different contexts distinguished into LowPowerContext and HighPowerContext and described by means of entries defining the device resources like CPU power, memory capacity, etc. In a service specification diagram, components can be annotated with QoS information like the meanTimeToFailure which is borrowed from [34] and which is an estimate of the mean time until a component's first failure. These information will be taken into account for non-functional analysis purposes.

Once the components implementing the eHealth service have been identified, their interactions can be specified. The *elementary service dynamics diagram* in Figure 10 depicts the message exchanges that occur when the notify operation of the eHealth service is invoked to answer the patient that pressed the alarm button. The notification exploits the execQuery() operation to interact with the DB component, and the setPort() operation to open a connection with the patient. The model contains also non-functional annotations in order to enable performance analysis. In particular, taking into account the UML SPT profile, the message invocations are PAStep stereotyped and tagged with additional information describing the resources required for their execution. For instance, the execution of the setPort() operation can require more CPU in case more queries have to be performed on the data base.

In order to have a comprehensive description of eHealth, each component implementing the service need to be specified at a lower level of abstraction by means of *component design diagrams* like the ones in Figure 11. In these diagrams, the classes implementing the DB and LOGIC components are modelled. As it will be clear later, the stereotype AdaptableClass is used to distinguish the classes which are adaptable according to specified resource annotations. For instance, the LOGIC component is implemented by means of the adaptable class NetworkStub, among others, which embodies a variability based on the resource NetworkInterface. The variability can be solved leading to the WIFI and Bluetooth alternatives with respect to the possible device contexts described in Figure 9. Component implementations can also depend on non-adaptable classes (e.g. the class implementing the DB component in Figure 11). This is the case of standard components that do not require resource adaptations.

To identify the physical mobile entities in the system we introduce some others modeling constructs like the <<PHMobEntity>> stereotype (see Doctor or Relative actor in the use case of Figure 5). This stereotype has a tag value (PHMpattern) that points to the mobility pattern of the identified entity. Such mobility pattern is described by a *physical mobility pattern diagram* that is a *state diagram* where the states represent the system configurations (hardware plus software) the entity interacts with during his/its moves, while the arrows model the feasible moving among configurations. Each state is annotated with: (i) the name of the deployment diagram describing the configuration represented by the

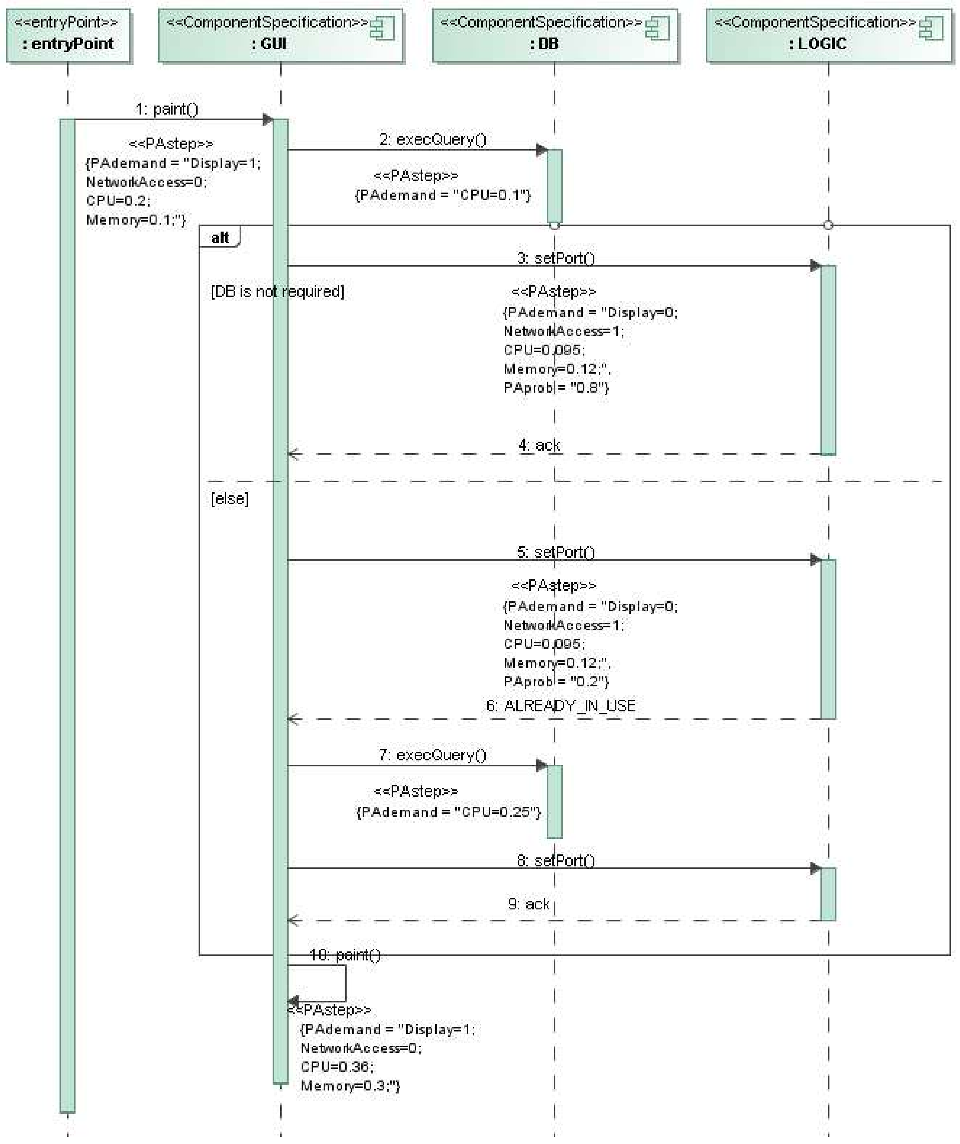
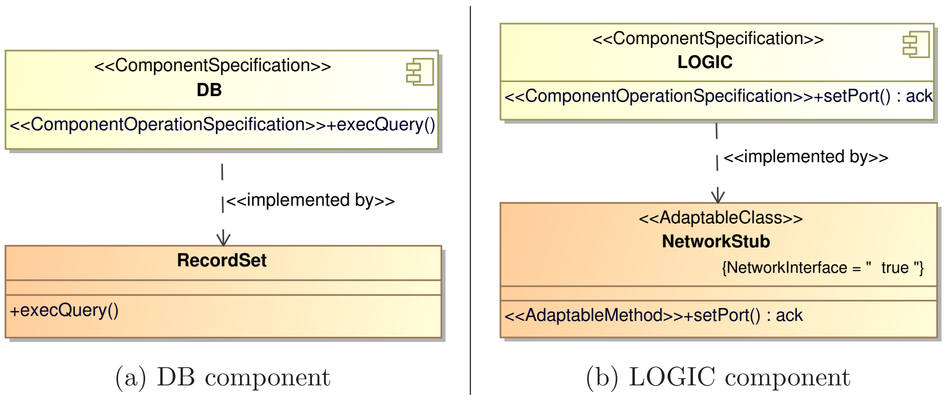

<<entryPoint>>
: entryPoint
<<ComponentSpecification>>
: GUI
<<ComponentSpecification>>
: DB
<<ComponentSpecification>>
: LOGIC
1: paint()
<<PAstep>>
{PAdemand = "Display=1; NetworkAccess=0; CPU=0.2; Memory=0.1;"}
2: execQuery()
<<PAstep>>
{PAdemand = "CPU=0.1"}
alt
3: setPort()
[DB is not required]
<<PAstep>>
{PAdemand = "Display=0; NetworkAccess=1; CPU=0.095; Memory=0.12;", PAprob = "0.8"}
4: ack
[else]
5: setPort()
<<PAstep>>
{PAdemand = "Display=0; NetworkAccess=1; CPU=0.095; Memory=0.12;", PAprob = "0.2"}
6: ALREADY_IN_USE
7: execQuery()
<<PAstep>>
{PAdemand = "CPU=0.25"}
8: setPort()
9: ack
10: paint()
<<PAstep>>
{PAdemand = "Display=1; NetworkAccess=0; CPU=0.36; Memory=0.3;"}

FIGURE 10. Elementary Service Dynamics Diagram

state (`PHMobContextInstanceRef` tag value), and (ii) the estimated total time the entity spends in the context during the observation interval (`PHMobRestime`).

In Figure 12.a we report the mobility pattern for the doctor in the eHealth service. At home the doctor activates his device and the eHealth service starts its


<<ComponentSpecification>>
DB
<<ComponentOperationSpecification>>+execQuery()
<<implemented by>>
RecordSet
+execQuery()
<<ComponentSpecification>>
LOGIC
<<ComponentOperationSpecification>>+setPort() : ack
<<implemented by>>
<<AdaptableClass>>
NetworkStub
{NetworkInterface = " true "}
<<AdaptableMethod>>+setPort() : ack

(a) DB component (b) LOGIC component

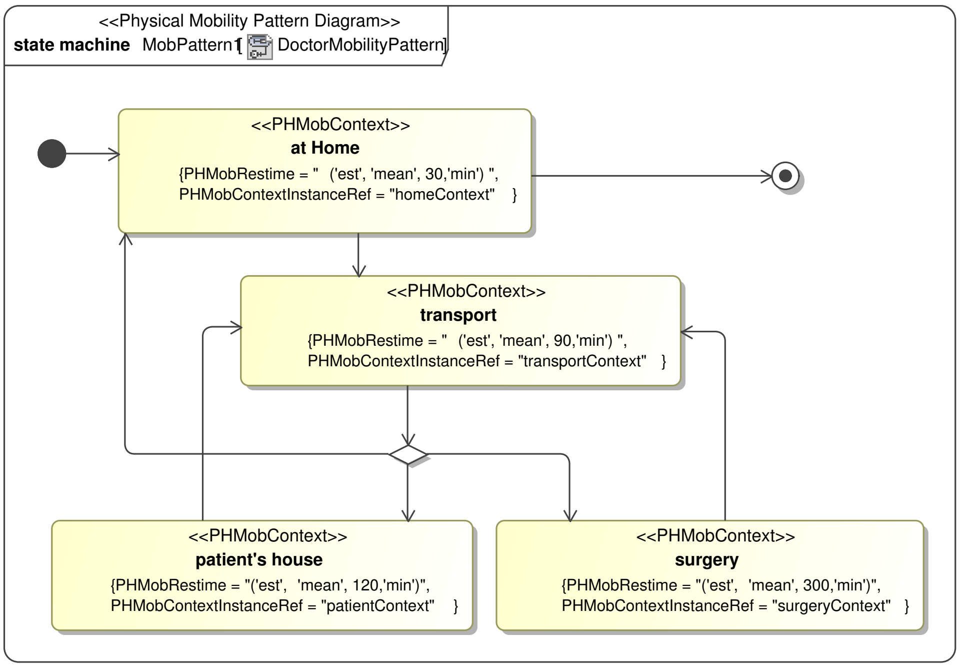
FIGURE 11. Component Design Diagrams

execution. During the working time, the doctor moves between the surgery and the houses of patients requiring assistance. The doctor service should be always available during the working time of the doctor and this implies an additional configuration that we model by means of the *transport* state.

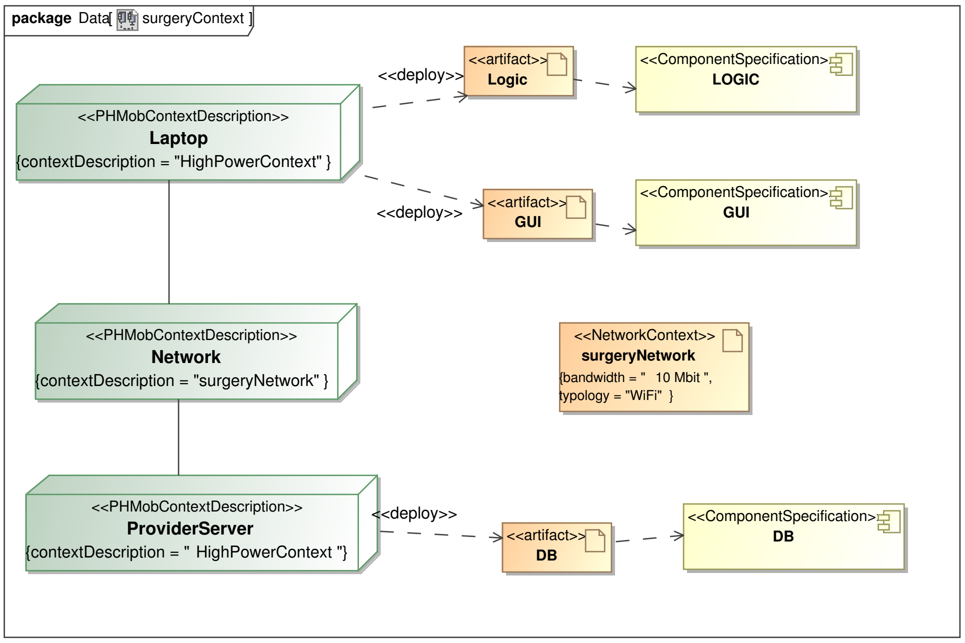
The system configurations (or contexts) are described by deployment diagrams. One deployment diagram is associated to each different physical context. Nodes in the deployment are stereotyped with `PHMobContextDescription` having a tag value (`ContextDescription`) pointing to the description of the device/network characteristics represented by the node. In Figure 12.b we report the configuration of the eHealth service for the doctor when s/he is in the surgery. The doctor runs the service (actually two components implementing the service) on his laptop and the available network connection is a WiFi network. The characteristics of the laptop and the provider server are described in the *HighPowerContext* description (see Figure 9) while the network features are described in the `NetworkContext` description reported for simplicity in the deployment diagram itself.

The last phase of the development process focuses on the code derivation of the resource-aware adaptable components implementing the modeled `eHealth` service. Model-to-code transformations are used for this purpose. In particular, relying on the CHAMELEON programming model (see below), models are translated automatically into generic code skeletons (like the one in Listing 1) by means of a developed code generator based on the *Eclipse Java Emitter Template framework* (part of the EMF framework [15]). JSP-like templates define explicitly the code structure and get the data they need from the UML model of the specified service exported into EMF. With this generation engine, the generated code can be customized and then re-generated without losing already defined customizations.

```
adaptable public class NetworkStub {
   ...
   adaptable public void NetworkStub();
      void setPort() {
```


<<Physical Mobility Pattern Diagram>>
state machine MobPattern1[DoctorMobilityPattern]
<<PHMobContext>>
at Home
{PHMobRestime = " ('est', 'mean', 30,'min') ",
PHMobContextInstanceRef = "homeContext" }
<<PHMobContext>>
transport
{PHMobRestime = " ('est', 'mean', 90,'min') ",
PHMobContextInstanceRef = "transportContext" }
<<PHMobContext>>
patient's house
{PHMobRestime = "('est', 'mean', 120,'min')",
PHMobContextInstanceRef = "patientContext" }
<<PHMobContext>>
surgery
{PHMobRestime = "('est', 'mean', 300,'min')",
PHMobContextInstanceRef = "surgeryContext" }

(a) Mobility Pattern


package Data[surgeryContext]
<<PHMobContextDescription>>
Laptop
{contextDescription = "HighPowerContext" }
<<deploy>>
<<artifact>>
Logic
<<ComponentSpecification>>
LOGIC
<<deploy>>
<<artifact>>
GUI
<<ComponentSpecification>>
GUI
<<PHMobContextDescription>>
Network
{contextDescription = "surgeryNetwork" }
<<NetworkContext>>
surgeryNetwork
{bandwidth = " 10 Mbit ",
typology = "WiFi" }
<<PHMobContextDescription>>
ProviderServer
{contextDescription = " HighPowerContext "}
<<deploy>>
<<artifact>>
DB
<<ComponentSpecification>>
DB

(b) Configuration

FIGURE 12. Mobility Pattern Diagram

```
        */ Method definition */
      }
   }

   alternative A1 adapts NetworkStub {
      void setPort() {
         */ Method definition that takes into account the WIFI resource*/
      }
   }

   alternative A2 adapts NetworkStub {
      void setPort() {
         */ Method definition that takes into account the WIFI resource */
      }
   }
}
```

Listing 1. Fragment of the generated *NetworkStub* adaptable class

With respect to service context-awareness and adaptability, a key role is played by the integration between analysis techniques (from which SLS is devised/refined) and the Model-to-code techniques. That is, the SLS (or an its part) drives the skeleton code generation. After the generation of the skeleton code has been done, the specific service logic within the generic code has to be implemented still taking into account the SLS. This is done by using the Chameleon Development Environment (and hence the Chameleon Programming Model) we are going to present. Finally, the whole code is validated against the devised SLS and hence certified.

In the following we briefly describe the Chameleon framework and how it supports the implementation of PLASTIC adaptive services and their provision and consumption.

▶ **Programming Model**. Referring to right-hand side of Figure 13, the *Development Environment* (DE) is based on a *Programming Model* that provides 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 13 the screen-shoot of our DE implemented as an Eclipse plugin [5]. 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) (as part of the PLASTIC registry).

▶ **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.

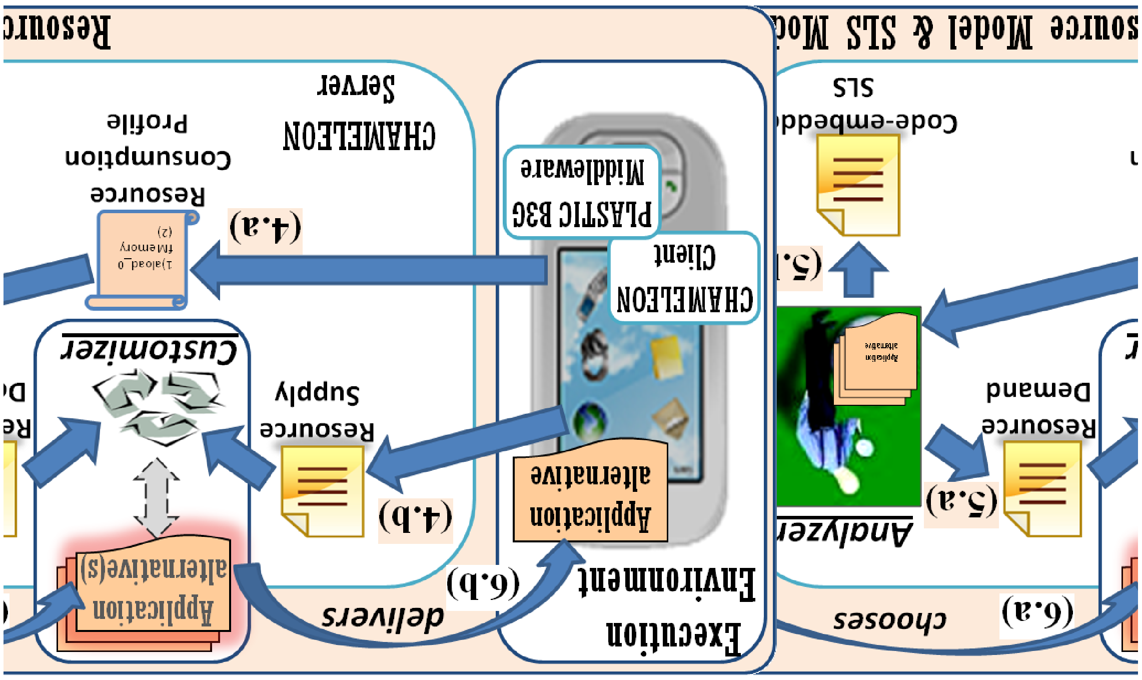


FIGURE 13. CHAMELEON Framework

▶ **Chameleon Server-side**. Still referring to Figure 13, 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). 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.

▶ **Chameleon Client-side**. *PLASTIC-enabled devices* (see left-hand side of Figure 13) are devices deploying and running the CHAMELEON *Client* component and the *PLASTIC B3G Middleware* [18] 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 the resource consumption profile.

▶ **Customizer**. 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 (via the Over-The-Air (OTA) provisioning technique [1]) 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.

4.2. The CONNECT project

Our everyday activities are increasingly dependent upon the assistance of digital systems that pervade our living environment. However, the current ubiquity of digital systems is technology-dependent. The efficacy of integrating and composing networked systems is proportional to the level of interoperability of the systems' respective underlying technologies. This leads to a landscape of technological islands of networked systems, although interoperability bridges may possibly be deployed among them. Further, the fast pace at which technology evolves at all abstraction layers increasingly challenges the lifetime of networked systems in the digital environment.

The CONNECT project [21] aims at dropping the heterogeneity barriers that prevent networked systems from being eternal, thus enabling the continuous composition of networked systems to respond to the evolution of functionalities provided to and/or required from the networked environment, independently of the embedded software technologies. CONNECT specifically targets the dynamic synthesis of connectors via which networked systems communicate. The resulting *emergent connectors* (or CONNECTors) then compose and further adapt the interaction protocols run by the connected systems, which realize application- down to middleware-layer protocols.

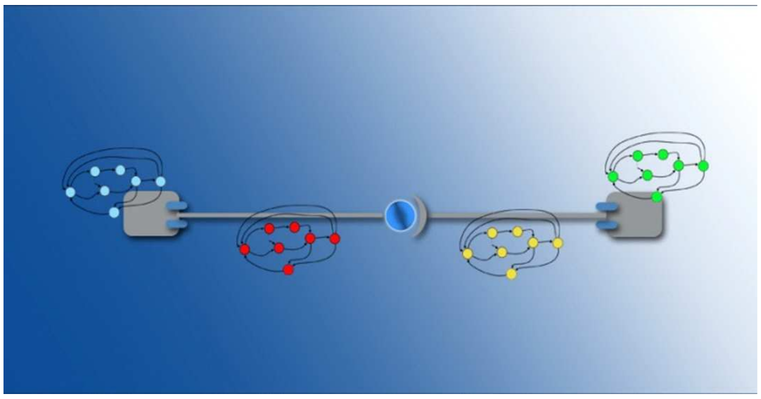


FIGURE 14. CONNECTors

The CONNECT synthesis process relies on a formal foundation for connectors, which allows learning, reasoning about and adapting the interaction behavior of networked systems. With respect to PLASTIC and PFM below (even though towards completely different goals), CONNECT operates a drastic shift by learning, reasoning about and synthesizing connector behavior *at run time*. Indeed, the use of connector specifications pioneered by the software architecture research field has mainly been considered as a design-time concern, for which automated reasoning is now getting practical even if limitations remain. On the other hand, recent effort in the semantic Web domain brings ontology-based semantic knowledge and reasoning at run time but networked system solutions based thereupon are currently mainly focused on the functional behavior of networked systems, with few attempts to capture their interaction behavior as well as non-functional properties. After the first year, the approach we have planned to undertake within CONNECT aims at learning the interaction protocols (both application and middleware layer) behavior by observing the interactions of the networked systems and the corresponding models, i.e, extended LTSs (see Figure 14), are derived and exploited at run time for generating connectors on the fly.

Towards this aim, CONNECT raises a set of unique challenges in the area of software systems engineering, from theoretical foundations to specify the interaction behavior of networked systems to run time methods and tools to turn specifications into running protocols, and vice versa. CONNECT addresses several key challenges: (i) interoperability that increasingly needs to be overcome by networked systems. This calls for a paradigm shift that goes beyond today's middleware solutions and effectively lies in the dynamic synthesis of emergent connectors. (ii) Theories for emergent connectors, (iii) dynamic connector behavior learning and (iv) synthesis, and (v) ensuring dependability of the overall synthesis process.

In the following we describes our early experience in the dynamic connector synthesis. With the aim of achieving eternal universal interoperability among heterogeneous networked systems, a key concept of CONNECT is synthesizing new interaction behaviors out of the ones implemented by the systems to be made interoperable, and further generating corresponding CONNECTors to bridge protocols. So far, we have devised a preliminary theory of *mediating connectors* (also called *mediators*).

The idea is to formally characterize mediators between mismatching protocols by rigorously defining, in particular, the necessary conditions that must hold for protocols to be mediated. These conditions will led to the definition of complex protocol matching and mapping relationships. The relationships represent two essential operations for the dynamic synthesis of mediating connectors to enable eternal networked systems. In fact, the matching relationship allows the rigorous characterization of the conditions that must hold for two heterogeneous protocols in order to interoperate through a mediator connector. Thus it allows one to state/check the existence of a mediator for two heterogeneous protocols. The mapping relationship allows the formal definition of the algorithm that should be performed in order to automatically synthesize the required mediator. Both relationships will be defined with respect to a behavioral model of the networked systems' interaction protocol. Thus, also in CONNECT, managing models at run time during the (synthesis) process is a key aspect and models play a key role in synthesizing the connector behavior and in validating that, through it, the desired system dependability is achieved.

Overview of the automated CONNECTor synthesis approach. Starting from two protocols, we want to check if their functionalities match, i.e., if the interacting parties may coordinate and achieve their respective goals. If this is the case, then we synthesize a mediator, otherwise they cannot communicate, at least based on our methodology. Figure 15 depicts the overall idea.

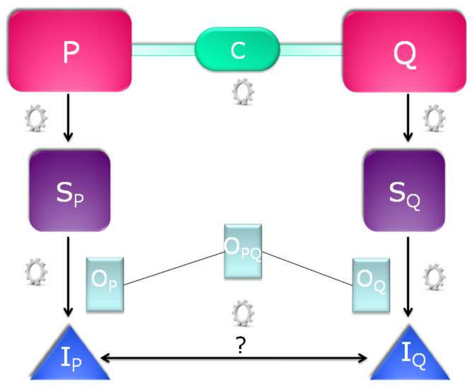


FIGURE 15. CONNECTor synthesis overview

The basic ingredients are: (i) the behavior of two protocols represented by LTSs P and Q, (ii) two ontologies O_P and O_Q describing the meaning of P and Q actions, and (iii) a mapping O_{PQ} between the two ontologies. Note that, when referring to protocol behavior, we mean the actions of a networked system that are observable at the interface level, i.e., its input/output actions. We further consider protocols P and Q that are minimal where we recall that every finite LTS has a unique minimal representative. Based on the structural characteristics of the two protocols, we build an abstraction for each of them, which we call structure. For P and Q, the abstraction is identified in the figure by S_P and S_Q, respectively. Then, using the ontology mapping function, we find the common language for the two protocols (pairs of words with the same meaning). This leads us to highlight the induced LTSs for both protocols (see I_P and I_Q), i.e., the structures where only the words belonging to the common language are highlighted. Finally, we check if the induced LTSs have a functional matching relation. In other words, we check if part of the provided/required functionalities of the two protocols are similar, i.e., are equivalent according to a suitable functional matching relation that we briefly describe below. If this is the case, then we synthesize a mediator, otherwise we cannot provide a mediator to let them communicate. Given two protocols P and Q, the mediator C that we synthesize is such that when building the parallel composition $P|Q|C$, the protocols P, Q are able to communicate to evolve to their final states. This is achievable by checking that the observable behaviors of P, Q are equivalent through a suitable notion of bisimilarity. The following briefly describes our protocol matching and mapping relationships. The former allows for checking whether there exists a mediator that makes two heterogeneous protocol interoperate. The latter allows for the definition of the algorithm that has to be performed in order to automatically synthesize the mediator.

CONNECT Matching Relationship. It defines necessary conditions that must hold in order for a set of networked systems to interoperate through a mediating CONNECTor. The matching condition is that they have complementary behaviors. Moreover, two functionalities are complementary if they can be abstracted by the same model under a suitable notion of behavioral equivalence that is driven by ontological information. If the functionalities of two networked systems match and, hence, the two networked systems perform complementary functionalities, then they can interoperate via a suitable mediating connector. In other words, if the matching relationship is satisfied, then there exists a mediating connector making the two networked systems interoperate. This mediating connector is abstracted by a mapping relationship. The synthesised mediator provides complementary interfaces to the two protocols, and manages exchange of information between the two (at the right synchronization point) driven by the mapping relationship.

4.3. The PFM project

A desirable characteristic of software systems, in general, is the ability to continue to properly work even after damages and system misbehaviors are experienced. This ability should allow for repairing the system configuration and maintaining a certain level of quality of service (e.g., performance and availability). To this aim, a crucial role is played by the ability of the system to react and adapt automatically (i.e., without the intervention of a human administrator) and dynamically (i.e., without service interruption, or with a minimal one).

This section briefly discusses PFM, a framework to manage performance of software system at run time based on monitoring and model-based performance evaluation [17]. The approach makes use of performance models as abstractions of the managed application that, although specified at design time, are also available at run time when the system is operating. The framework monitors the performance of the application and, when a performance problem occurs, it decides the new application configuration on the basis of feedback provided by the on-line evaluation of performance models. The main characteristic of this approach is the way reconfiguration alternatives are generated. In fact, differently from other approaches, PFM does not rely on a fixed repository of predefined configurations, rather, starting from the data retrieved by the on-line monitoring of the performance models (that represents a snapshot of the current "performance state"), a number of new configuration alternatives are dynamically generated by applying the rules defined within the reconfiguration policy. Once such alternatives have been generated the on-line evaluation is performed by predicting which one of them is most suitable for resolving the problem occurred. In particular, the choice of the new system configuration might consider several factors, such as, for example, security and reliability of the application, and resources needed to implement the new configuration.

In this approach performance evaluation models are used to predict the system performance of the next system reconfiguration alternative. The performance models are "product form QNs". To this aim, each eligible system configuration is described by means of a predictive model instantiated with the actual values observed over the system until the moment of the performance alarm. The models are then evaluated and on the basis of the obtained results the framework decides the reconfiguration to perform over the software system. Therefore, the predictive models, representing the system alternatives, are evaluated at run time and this poses strong requirements on the models themselves. PMF has been experimented to manage the performance of the PFM publish/subscribe middleware [16, 19]. The experiment shows that the usage of predictive models improves the decision step. The system reconfigured according to the chosen alternative has better performance than the other alternatives generated during the reconfiguration process. The configuration alternatives we experimented all deal with structural changes of the PFM network topology in order to improve messages routing.

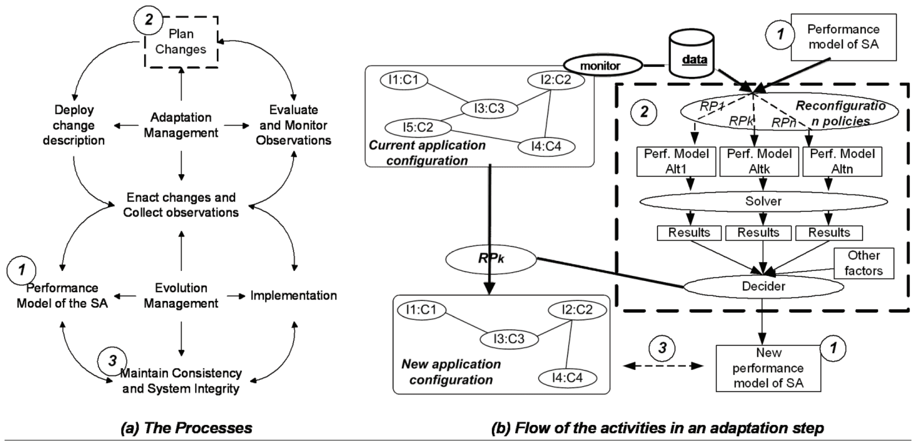

2
Plan Changes
Deploy change description
Adaptation Management
Evaluate and Monitor Observations
Enact changes and Collect observations
1
Performance Model of the SA
Evolution Management
Implementation
3
Maintain Consistency and System Integrity
(a) The Processes
monitor
data
1
Performance model of SA
I1:C1
I2:C2
I3:C3
I5:C2
I4:C4
Current application configuration
2
RP1
RPk
RPn
Reconfiguration policies
Perf. Model Alt1
Perf. Model Altk
Perf. Model Altn
Solver
Results
Results
Results
Other factors
RPk
Decider
I1:C1
I2:C2
I3:C3
I4:C4
New application configuration
3
New performance model of SA
1
(b) Flow of the activities in an adaptation step

Figure 16. Adaptation for performance

Figures 16(a) and 16(b) describe the processes and flow of the activities in an adaptation step of the PFM framework, respectively. In particular, with reference to Figure 16(a), Figure 16(b) outlines a reconfiguration loop exploding the Plan

`Changes` step (see label 2). The application configuration is modelled by means of a performance model at the software architecture level (see label 1). During the monitoring the framework observes a performance problem and throws an alarm. After the analysis of the monitored data, the `plan changes` activity starts. Inputs of this activity are the performance model of the current running system configuration and the monitored data. Given the set of reconfiguration policies RP_i defined for the application, a set of suitable reconfiguration alternatives is generated. Each alternative is modelled by means of a performance model, created on-the-fly by modifying the current performance model. Each model is initialized with the monitored data, and then evaluated by using a solver tool. Finally, the evaluation results are compared, and the most rewarding configuration is selected and applied to the system. This step (step labeled with 3) maintains by construction the consistency among the (new) performance model and the system configuration.

5. Concluding remarks

In this paper we have discussed our thesis on software in the future. Softure requires to rethink the whole software engineering process since it never stabilizes, but it is permanently under maintenance. Software evolution, which is traditionally practiced as an off-line activity, must often be accommodated at run time for Softure.

Run time models, languages and methodologies will play a key role in achieving adaptability and dependability for future software applications. In a broader software engineering perspective it is therefore mandatory to reconcile the static/compile time development approach to the dynamic/interpreter oriented one, thus making models and validation technique manageable lightweight tools for run time use. There are several challenges that are far to be solved in this domain.

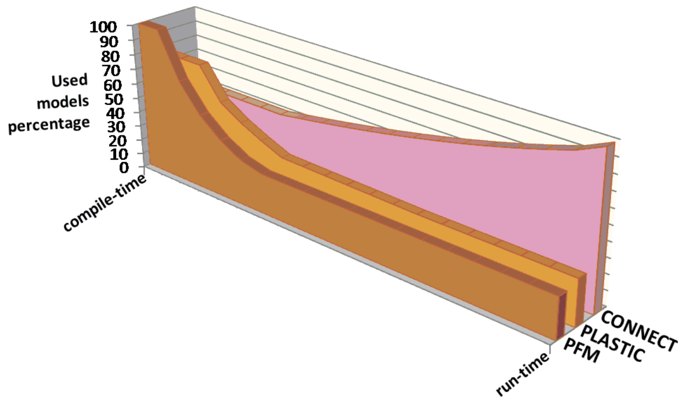


FIGURE 17. Models@run.time degree

We reported three experiences to adaptive software development and provision as different instances of the problem raised by Softure. These experiences represent our attempts towards the non trivial concrete instantiation of the Softure development process. They are not entirely innovative *per se* rather they are used in a completely new and non trivial fashion. The three experienced approaches are different in nature with respect to the usage degree of models at run time. In Figure 17, we informally compare the three approaches by showing the tendency of models usage along the software life cycle. From compile to run time, the three diagrams show the percentage of the total number of models used by each approach. This dimension gives us a means to evaluate the underlying development processes with respect to the requirements dictated by Softure. The PFM, PLASTIC, and CONNECT development processes are *softure development processes* since they all allow for the construction of self-adaptive systems that manage models at run time in order to adapt to possible changes. In PFM, all the models are built at design-time, and those needed for performance prediction and dynamic reconfiguration are kept at run time. In PLASTIC, not all the models are built at design-time. Part of them plus new ones, generated during later phases, are kept at run time in order to perform context-aware and SLS-based adaptation. In CONNECT, only ontological information and the related models are available at design-time. By exploiting learning and synthesis techniques, all the needed models are automatically inferred and used at run time in order to achieve dynamic interoperability. It is worth noting that, due to a higher percentage of models managed at run time, in CONNECT the efficiency of the dynamic model-based analysis is more crucial.

Summarizing our message is that in the Softure domain it is important to think and research *point to point* theories and techniques but it is mandatory to re-think the whole development process breaking the traditional division among development phases by moving some activities from design-time to deployment- and run time, hence asking for both more efficient verification and validation techniques and new modeling notations and tools. Furthermore, to support dynamic adaptation, the focus of software development should shift from a traditional and static approach where most of code is written from scratch to a more reuse-oriented and dynamic approach where the needed components or services are first discovered and then assembled together to form the system to be executed. This lead to develop support for an evolutionary development process made of continuous *discover-assemble-execute* iterations. Development methodologies and tools must account in a rigorous way of *quantitative* concerns, allowing programmers to deal with these concerns declaratively. Models must become simpler and lighter by exploiting compositionality and partial evaluation techniques. Innovative development processes should be defined to properly reflect these new concerns arising from software for ubiquitous computing.

References

[1] Over-The-Air (OTA). http://developers.sun.com/mobility/midp/articles/ota/.

[2] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in archjava. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 334–367, London, UK, 2002. Springer-Verlag.

[3] J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.

[4] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[5] M. Autili, P. D. Benedetto, and P. Inverardi. CHAMELEON project - SEA group. http://di.univaq.it/chameleon/.

[6] M. Autili, P. D. Benedetto, and P. Inverardi. CHAMELEON project - SEA group. http://sourceforge.net/projects/uda-chameleon/.

[7] M. Autili, P. D. Benedetto, and P. Inverardi. Context-aware adaptive services: The plastic approach. In M. Chechik and M. Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 124–139. Springer, 2009.

[8] M. Autili, L. Berardinelli, V. Cortellessa, A. Di Marco, D. Di Ruscio, P. Inverardi, and M. Tivoli. A development process for self-adapting service oriented applications. In *Proc. of ICSOC*, Vienna, Austria, Sept 2007.

[9] M. Baldauf, S. Dustdar, and F. Rosenberg. A survey on context&#45;aware systems. *Int. J. Ad Hoc Ubiquitous Comput.*, 2(4):263–277, 2007.

[10] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE TSE*, 30(5):295–310, 2004.

[11] L. Baresi, R. Heckel, S. Thone, and D. Varro'. Style-based refinement of dynamic software architectures. In *WICSA '04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture*, page 155, Washington, DC, USA, 2004. IEEE Computer Society.

[12] G. Barthe. MOBIUS, Securing the Next Generation of Java-Based Global Computers. ERCIM News, 2005.

[13] A. Bertolino, G. D. Angelis, A. Di Marco, P. Inverardi, A. Sabetta, and M. Tivoli. A Framework for Analyzing and Testing the Performance of Software Services. In *In Proc. of the 3rd ISoLA*, volume 17. Springer, 2008.

[14] A. Bucchiarone, P. Pelliccione, C. Vattani, and O. Runge. Self-repairing systems modeling and verification using agg. In *WICSA/ECSA*, pages 181–190, 2009.

[15] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework*. Addison Wesley, 2003.

[16] M. Caporuscio, A. Carzaniga, and A. Wolf. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *Software Engineering, IEEE Transactions on*, 29(12):1059–1071, Dec. 2003.

[17] M. Caporuscio, A. Di Marco, and P. Inverardi. Model-based system reconfiguration for dynamic performance management. *J. Syst. Softw.*, 80(4):455–473, 2007.

[18] M. Caporuscio, P.-G. Raverdy, H. Moungla, and V. Issarny. ubiSOAP: A service oriented middleware for seamless networking. In *Proc. of 6th ICSOC*, 2008.

[19] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *Foundations of Intrusion Tolerant Systems*, 0:283, 2003.

[20] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors. *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, volume 5525 of *Lecture Notes in Computer Science*. Springer, 2009.

[21] CONNECT FET Project. Home page on line at: http://connect-forever.eu/.

[22] H. Ehrig, C. Ermel, O. Runge, A. Bucchiarone, and P. Pelliccione. Formal analysis and verification of self-healing systems. In *FASE*, pages 139–153, 2010.

[23] I. Georgiadis, J. Magee, and J. Kramer. Self-organising software architectures for distributed systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 33–38, New York, NY, USA, 2002. ACM.

[24] V. Grassi, R. Mirandola, and A. Sabetta. A model-driven approach to performability analysis of dynamically reconfigurable component-based systems. In *WOSP*, pages 103–114, 2007.

[25] D. Hirsch, P. Inverardi, and U. Montanari. Graph grammars and constraint solving for software architecture styles. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 69–72, New York, NY, USA, 1998. ACM.

[26] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[27] IFIP WG 10.4 on Dependable Computing and Fault Tolerance: http://www.dependability.org/wg10.4/.

[28] P. Inverardi and M. Tivoli. The future of software: Adaptation and dependability. pages 1–31, 2009.

[29] J. Skene, D. Lamanna, W. Emmerich. Precise service level agreements. In *Proc. of the 26th ICSE*, pages 179–188, Edinburgh, UK, May 2004.

[30] D. Le Metayer. Describing software architecture styles using graph grammars. *Software Engineering, IEEE Transactions on*, 24(7):521–533, Jul 1998.

[31] M. Autili, P. Di Benedetto, P. Inverardi, D. A. Tamburri. Towards self-evolving context-aware services. In *Proc. of Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS), DisCoTec'08*, volume 11, 2008. http://eceasst.cs.tu-berlin.de/index.php/eceasst/issue/view/18.

[32] J. Magee and J. Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, 1996.

[33] G. C. Necula. Proof-carrying code. pages 106–119. ACM Press, 1997.

[34] OMG. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, 2006. Version 1.0, formal/06-05-02.

[35] OMG. UML Profile for Modeling and Analysis of Real-Time and Embedded systems, 2009. http://www.omg.org/spec/MARTE/1.0/.

[36] PLASTIC IST Project. Home page on line at: http://www.ist-plastic.org.

[37] F. Raimondi, J. Skene, and W. Emmerich. Efficient Online Monitoring of Web-Service SLAs. In *Proc. of the 16th ACM SIGSOFT/FSE*, Nov. 2008.

[38] B. Schilit, N. Adams, and R. Want. Context-aware computing applications. In *IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, CA, US*, 1994.

[39] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 179–193, London, UK, 2000. Springer-Verlag.

[40] J. yi Hong, E. ho Suh, and S.-J. Kim. Context-aware systems: A literature review and classification. *Expert Systems with Applications*, 36(4):8509 – 8522, 2009.

[41] T. Zahariadis and B. Doshi. Applications and services for the B3G/4G era. *Wireless Commun., IEEE*, 11(5), Oct 2004.

Marco Autili
Dipartimento di Informatica - Università degli Studi di L'Aquila, Italy
e-mail: `marco.autili@di.univaq.it`

Paola Inverardi
Dipartimento di Informatica - Università degli Studi di L'Aquila, Italy
e-mail: `paola.inverardi@di.univaq.it`

Massimo Tivoli
Dipartimento di Informatica - Università degli Studi di L'Aquila, Italy
e-mail: `massimo.tivoli@di.univaq.it`