Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 189 (2007) 35–50
www.elsevier.com/locate/entcs

Autonomic Adaptation solution based on Service-Context Adequacy Determination

Marcel Cremene[1]

Dep. Communications
Technical University of Cluj-Napoca
Cluj-Napoca, Romania

Michel Riveill[2]

Lab. I3S, Eq. Rainbow, CNRS
Université de Nice
Sophia-Antipolis, France

Christian Martel[3]

Lab. SysCom
Université de Savoie
Chambery, France

Abstract

Autonomic adaptation is a very ambitious emerging domain aiming to build self-adaptable systems. The most important advantages of these systems are: easier complexity management, autonomous service evolution and proactive behaviour. In the 'classical' service adaptive systems, the developer must solve 'manually' two problems. The first one is to determine a priori the service adequacy to each possible context. The second one depends on the first one and it is to specify a strategy, a reconfiguration suite, which will transform an inadequate service into an adequate one. In order to replace the developer-based reasoning with a machine-based one, we propose a meta-model describing the service and its context into a common graph representation. A set of general rules and operators applied on this meta-model enables the machine to check the service adequacy to its context. The same meta-model is used for searching the adaptation strategy.

Keywords: Autonomic, Adaptation, Service, Context, Adequacy, Meta-Model

[1] Email: cremene@com.utcluj.ro
[2] Email: riveill@unice.fr
[3] Email: christian.martel@univ-savoie.fr

doi:10.1016/j.entcs.2007.05.046

1 Introduction

1.1 Background

We are living today in a world where the computing/telecommunications infrastructure and software services running on top of this infrastructure, are more and more ubiquitous and diverse. We consider a service as implemented by a distributed architecture composed by interconnected components (sometimes called also services). Reusable components may be published, upgraded and removed continuously by different providers.

We consider the service's *context* [5] as constituted by two main parts:

a. The *infrastructure related part* includes the available resources and the elements that may influence this infrastructure, for instance, the wireless network throughput may be influenced by the rain.

b. The *user related part* includes the user needs and the factors that may influence these needs, for instance, the proximity to some objects that may change the user interest/needs.

Dynamic service adaptation means to reconfigure a service in execution in order to make it more adequate to its context. Accordind to our service definition, the reconfiguration primitives are: add, replace, remove, parameterize, migrate components and connect/disconnect components ports. In the 'traditional' adaptive systems, the developer must solve 'manually' two main problems:

P1. *Determine, a priori, the service adequacy to each possible context state.* Usually, this problem is solved using some rules and conditions that test some context attributes.

P2. *Specify a strategy that will transform an inadequate service into an adequate one.* A strategy is a suite of reconfiguration primitives.

The second problem depends on the first one. Thus, in this paper we insist especially on this first problem and we propose some partial solutions for the second one.

1.2 Problem

The fact that the developer must solve 'manually' the problems described before leads to several drawbacks, as follows:

- *High complexity/costs.* The infrastructure, the services but also the user needs are in continuous diversification. The increasing *complexity* becomes an important problem for the developers/administrators, problem noticed also in [13]. If the context includes a large number of elements with many attributes and we have a services with a lot of components, the determination of service adequacy to context may lead to a combinatorial explosion. Additionally, the possible rule conflicts must be also solved by the developer.
- *Impossibility to completely anticipate the context/service evolution.* In open en-

vironments [7], such as mobile systems, it is impossible to anticipate all context elements/attributes that will influence the service at a given moment. Instead of having services that offer only the features anticipated by the service developer, we may prefer to have proactive services able to discover and propose to user new, unanticipated features. Also, a service that uses only components a priori defined cannot make use of new components that are published after its creation.

- *Difficult intervention a posteriori.* The developer intervention after the service creation may be impossible for some distributed services because they cannot be stopped (without some important costs) or the time available for reconfiguration is too short. In cases like this, it is useful to replace developer intervention by some automatic solutions.

1.3 General objective

The problems described before have a common cause: only the service developer understands if a service S is adequate to a given context C and decide what strategy to apply. Our objective is to replace the developer's work described before with an 'intelligent' algorithm. This algorithm will be able to determine itself the adequacy between the service and the context. Once this problem solved, the strategy search becomes possible because we can generate any service-context configuration and check its adequacy.

1.4 Approach

In order to make possible the machine-based reasoning about the service adequacy to the context, we propose a service-context meta-model. This meta-model use a common graph representation for describing: the service, the context and their interactions. The proposed meta-model will be used by the control part of an adaptation platform. The meta-model must be extended/updated dynamically as the context and service change, it must reflect at each moment the knowledge about the service and context.

1.5 Paper outline

This paper is organized as it follows: the next section presents several existent adaptive systems; section three presents the service-context meta-model, how it is used, and finally its place into adaptive platform architecture. Section four describes a simple prototype that we have implemented in order to test our model. The last section presents the evaluation, the conclusions and the future work.

2 Existent solutions, limitations and reusable aspects

In this section we analyse some representative adaptive platforms. Accordind to our objective, the main questions are: How is it determined the service adequacy to the context? How are specified the adaptation strategies? What is the developer role?

We are also interested to analyse if the adaptive platforms architectures are suitable for our objective and what are the elements of these architectures that may be reused. For avoiding possible confusions, in this paper we use 'service architecture' for describing a service and 'adaptive platform architecture' for describing the platform that adapts the service.

2.1 Adaptive platforms examples

An first interesting example is the "Rainbow" [10] platform proposed by D. Garlan et all. This platform architecture is based on the separation between the adapted part (the service) and the adaptation part (probes, control, effectors). The adaptation is controlled using rules and strategies. For instance, the rule indicates *when* a specific strategy must be applied and specify also what particular strategy to execute. A rule example is: "if the server response time is higher than a given value, then execute the strategy 'responseTimeStragey' ". This strategy specify *where* (the server group), and *what* to reconfigure (add a new server). The strategy includes itself some other rules (the separation rule/strategy is not very strong). A *addServer()* method must exist in order to make the adaptation possible. The service adequacy to the context is given by the rules; the adaptation solution is to apply a specific strategy. We can observe that, the rules and the strategies are: developer-made, predefined, and they are service and context specific. To take into account a new context element requires adding at least one rule or strategy. Determination of service adequacy to the context and strategy search, are developer-made tasks.

Some authors discuss about 'unanticipated adaptation'. According to J. Keeney [11], in a 'completely unanticipated system', the answers to questions such as: when, where, what and how to reconfigure a service are known only at runtime. The author proposes the "Chisel" platform that supports dynamic rule modification. Let analyze the following example:

```
ON WirelessNetworkDisconnect
NetworkConnectionService.WiredConnectionBehaviour
IF (NetworkConnectionService.WiredConnectionAvailable == TRUE &&
WirelessNetworkDisconnect.IsTemporary == FALSE ) ||
(UserPreferences.getPreferredComms()).compareTo("Wireless") != 0
```

This rule is that if the wireless network is disconnected, apply the strategy 'WiredConnectionBehaviour', but only if a wired connection exists and the wireless disconnection is not just temporary. The adaptation technique is based on reflective language support and will not be detailed here. We may observe that, even if these rules may be changed at runtime (as the author claims), the strategies are still predefined (i.e. 'WiredConnectionBehaviour' strategy) and also the events that comes from the context (i.e. 'WirelessNetworkDisconnect'). Another inconvenient is that, adding new rules require developer intervention because the user is not able in general to write such complex syntaxes or deal with possible conflicts between different rules. Basically, we encounter here the same limitations as for the "Rainbow" platform: the determination of service adequacy to its context is developer-based.

Autonomic Computing [13] is a very interesting emerging domain with high ambitions. IBM has an entire department working on this area. They have created the "IBM Autonomic Computing Toolkit" [12]. This toolkit comes with a general architecture inspired from the biological model: nerves-probes, brain-controller, and muscles-effectors. The reconfiguration decisions are based on the log messages but no general solution is given, the adaptation control is service specific. We have not found a solution allowing to automatically determine the service adequacy to its context. In fact, we do not have today a really complete solution for autonomous adaptation but rather several proposals around this subject.

K. Fujii proposes in [9] a semantic component model called 'CoSMoS' and a platform called 'SegSeC'. The proposed platform is able to automatically deploy component-based services, starting only from the user demand (which is expressed as natural language sentence). The main idea is to add semantic concepts to component models and then to use ontology and semantic graph in order to establish connections between components/services described by semantically related concepts. This idea is interesting because it provides some autonomy: the service is assembled without requiring developer intervention. However, some limitations of this platform are: the service is not reconfigured dynamically after the deployment, the only context taken into account is the user demand, the user demand must be simple enough and must respect a certain syntax (words order). This proposal still does not offer a general response to the problem of adequacy determination between a service and its context but it offers us a hint: to use additional semantic concepts.

After studying some other platforms such as: "MobiPADS" [3], "Gaia" [14], "K-component" [6], "Madam" [8] and others our conclusions are:

- *Platform architecture.* Existent adaptive platforms have in general three main parts: a) the adapted part (the service) having some reconfigurable elements (parameters), b) the observation part (probes, sensors) that periodically monitors the context and service state and c) the control part that uses rules and strategies. The most common principle used is the classical closed-loop, and it is similar to human nervous system model.
- *Control part.* The platforms control part is based on service and context-specific rules/strategies, which are predefined by the developer. Adding new context elements/attributes requires human intervention especially because not the platform but the developer only is able to reason about the service to context adequacy.

2.2 Platforms evaluation

In the analyzed platforms we have not found automatic solutions for the service-context adequacy determination. However, we may reuse some elements such as: the closed-loop principle for the adaptation platform, the adaptation techniques such as reflexive middleware, the context discovery solutions [4], the idea to add semantic information to component models. The only part of an adaptation platform that must be changed significantly in order to support autonomic adaptation is the control part.

3 Proposed solution for autonomic adaptation

This section is organized in a 'bottom-up' manner: we present first the service-context meta-model, its properties, its usage and finally the adaptive platform that uses this meta-model at the control level.

3.1 What is the service-context meta-model?

The service-context meta model describes the following aspects: S - *service*, I - *infrastructure*, IC - *infrastructure context*, U - *user(s)*, UC - *user's context*. These parts are connected like it is shown in the figure 1.

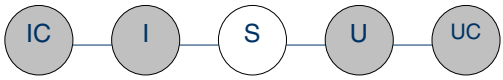

Fig. 1. Service-context meta-model, general structure

The service interacts directly with its infrastructure I and its user U. The infrastructure is influenced by its context IC and the user also is influenced by its context UC. According to context definition given in the introduction, the context C = {I, IC, U, UC}. We have two types of general interrelations:

- The *information exchange* relation concerns the S-U interaction. The user exchanges information with the service through the HMI (Human Machine Interface), the components exchange also information one with another.
- The *resources utilization* relation concerns the S-I interaction. The service uses a certain amount of the infrastructure resources (i.e. memory, bandwidth, etc.).

Service model

The service meta-model describes the service architecture as a graph having as nodes the components and as arcs the interconnections, similar to an ADL (Architecture Description Language) description. We observed that a "pipe-and-filter" perspective significantly simplifies our model without limiting it. A filter has one input and one output, a sink has only one input, and a source has only one output. A complex component with several inputs and several outputs will be represented, at the meta level, as decomposed in several distinct filters/sources/sinks.

Context model

The context is represented today using different model types: list of attributes, object oriented models, ontology based models[15], contextual graph [2]. In order to have a common model, we propose the use of the same model for the context and the service. This means, the context is also seen as an architecture composed by interconnected *contextual components*: user, terminal, network, environment, etc.

Profiles

One problem is that actual service and context models do not reveal the service-context interrelations. In order to solve it, we introduce the *profile* concept. The profile role is to describe the components and to show how they interact with each other. The profile is a meta-data (XML file) associated to a component, software or contextual. The profile elements that describe one filter are:

- *Type.* The type may be software, contextual or observer. An observer is a special software component used for service and context observation, it offers information about the others service/context components.
- *Component attributes.* The component attributes are related to the whole component. Examples of such attributes are: memory, CPU, APIs, OS, and they are related usually to the physical and logical resources. Each attribute has a name and a definition domain (values).
- *Flow attributes.* The flow attributes are related to the input and output information flows that enter or exit the component ports. These attributes characterize the information content. Some examples are: data type, language, compression, delay, bit rate. Each attribute has a name and a definition domain. For each attribute we define a *transfer function* **H**, as for the electrical filters. This function indicates the relationship between the output value and the input value for a certain attribute. The **H** function may have parameters that are associated with the component parameters. If the *H* relationship cannot be known a priori, it will be marked as unknown (we use the special symbol '?').

Each complex component may be decomposed in several filters. From the 'memory' attribute point of view, for instance, the terminal is seen as a memory source and the components are memory sinks. An English to french translation component is a memory sink but also a language filter. The component developer must always specify the component profile; otherwise the adaptation platform cannot be aware of the component capabilities.

Scenario and illustration

In order to facilitate the model explanation, we propose to apply it for a concrete scenario. This scenario is based on a forum service. The forum architecture is composed by a client, itself composed by a message editor *TreeViewUI* and a message composer *EditorUI*, and a forum server *Forum Server*. The service's infrastructure includes a smart-phone, which is the client host, connected through a wireless network to a server machine. Figure 2 illustrates the service-context model for this scenario.

Several contextual components are involved: *User* that interacts with the forum service through its HMI, *Terminal* that includes two components: the display as output device and the keyboard as input device. The terminal devices are interposed between the user and the software components. The terminal screen visibility may be influenced by external factors as the external light for instance. The *Network* connects the terminal with the internet access point and finally with the server

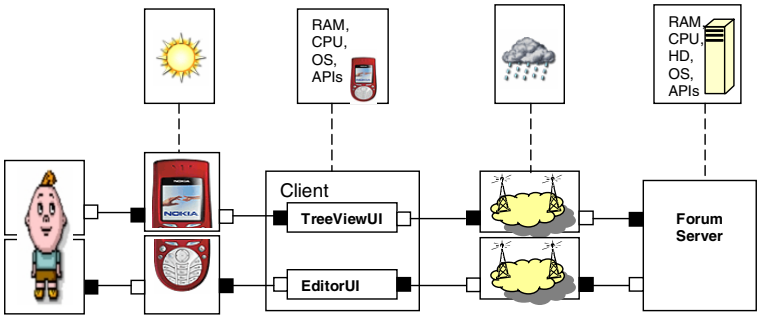

Fig. 2. Service-context model - architectural illustration

machine. Several networks may be concatenated. Uplink and downlink are seen as separated filters. The network bit rate may be influenced by the external conditions such as rain for instance. The *Server machine* is a host for the forum server (we may take into account this contextual component for load balancing).

Service-context meta-model represented as a graph

The adaptation platform manipulates the service-context meta-model as a graph. The graph depicted in figure 3 corresponds to the architecture depicted in figure 2. In order to simplify the graph, we have omitted some elements compared to the figure 2. The graph nodes corresponds to software, white, and contextual, gray components. The observer components are not included in this graph but we use them in order to update it. A node is an object that includes all the profile attributes: component attributes, flow attributes and transfer functions. The graph arcs are oriented and correspond either to information flows (normal arrows) or to resource utilization (dotted arrows).

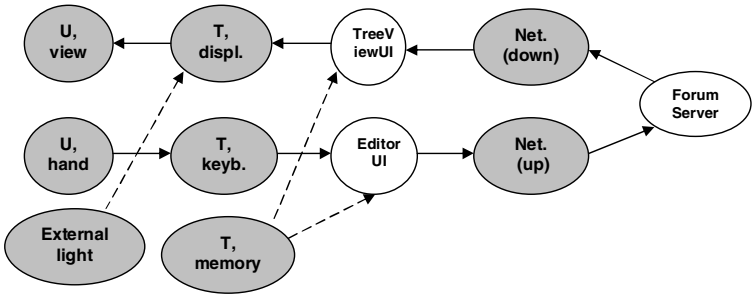

Fig. 3. Service-context model represented as a graph

3.2 How can we determine the service-context adequacy?

In order to formalize what 'adequacy' means, we consider that adequacy may be TRUE or FALSE. According to the interrelation types defined before, we express a general adequacy rule:

```
Adequacy = TRUE
```

```
IF
for any two interconnected filters Ci -> Cj, the Ci output value
is included in the Cj input interval
AND
for each type of consumed resource, the provided resource
of the same type is superior as value
```

Informational flow-related adequacy

The first part of the rule concerns the informational flow compatibility and the second the infrastructure resource compatibility. For the forum service, if the user (contextual component) produces information having the 'language' attribute 'FR', but the service input requires the value 'EN' for the same 'language' attribute, we decide that this service is not adequate to its context. Figure 4 illustrates this idea of information flow compatibility.

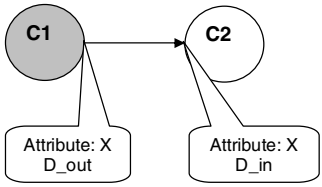

Fig. 4. Flow compatibility

The information flow circulates directly through the interconnections (but may suffer modification through the filters). For each attribute, the chain effect is given by the mathematical composition of the transfer functions for each filter in the chain (functions **H** specified in the component profile). If a filter profile does not specify an attribute, we suppose that the filter does not affect that attribute and its output value is equal to its input value. An unspecified function **H** is equivalent to the identity function, and this allows us to compose the functions. As it is depicted

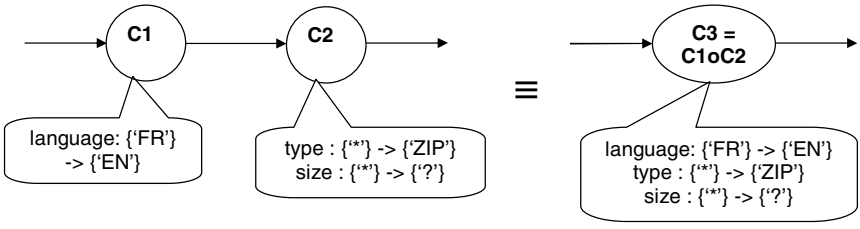

Fig. 5. Component chain composition

in 5, a chain made of two components C1 (a translator) and C2 (an archiver) is equivalent to a component C3 = C1 o C2 (composition). The C3 profile may be calculated automatically and contains all the attributes of the C1 and C2 profiles. The symbol '*' means any value and the symbol '?' means unknown value. The unknown values may be detected using specialized components, the observers, (i.e. language detector). The chain flow attributes are given by the reunion of all flow attributes of each filter. If two filters affect the same attribute (fact indicated by

their profiles), we use the standard mathematical function composition in order to determine the chain global transfer function. The information composition is analog to signal composition for electrical filter chains.

Infrastructure-related adequacy

A service is adequate to its context (physical infrastructure: terminals, machines, networks) if it does not need more resources than those available. The nature of the resources is usually additive. For instance, if the *TreeViewUI* (see figure 3) takes 100KB memory and *EditorUI* takes 300KB, then the two components will need at least 400KB. As we can see in 3), these two components consume the terminal memory, *T.memory*. If the available terminal memory is lower than 400KB, we have an inadequacy.

In some cases, the resource utilization does not fit into a simple additive relation. For instance, two visual components may be displayed on the same display at the same time or consecutively. The required screen surface is different in the two situations. In order to enable the developer to express different situations, we offer the possibility of using different composition and validation operators, not only the addition and the inferiority operators (see figure 7). The attribute definitions and the operators must be shared by all component developers in order to be able to integrate components produced by different developers.

3.3 How can we find the right adaptation strategy?

To find an adaptation strategy is to establish a list of reconfiguration primitives that will be applied. Two types of decisions are necessary: a) decide the reconfiguration type (replacement, insertion, removal, migration, parametrisation) and b) decide what component/interconnection to reconfigure into the existent service-context system. The proposed meta-model does not solve the strategy search problem but it allows us to generate different service-context configurations and check their adequacy. We present below a simple example demonstrating how the meta-model helps us to find a strategy.

An example

If the inadequacy is caused by information flow incompatibility, one general solution is to insert an additional filter that acts like an adapter. A translator is an adapter between two components (software or contextual) "speaking" different languages. The necessary component is searched for by its profile: the input/output function of the searched component must solve the inadequacy problem.

The insertion point is searched by verifying the profiles and the syntactical interface (IDL) compatibility. As we see in the figure 6, the graph model gives us the possibility to follow the informational flows. In the forum case, a new language translator filter may be inserted after the *EditorUI* output, if present on the terminal, or before the *ForumServer* input if the translator is a remote web service. The same idea may be used for the service output: a second translation component may

be inserted after the *ForumServer* output. As we can see, the model enables us to make distinction between the user's ability to write or to read respectively (input and output) in English.

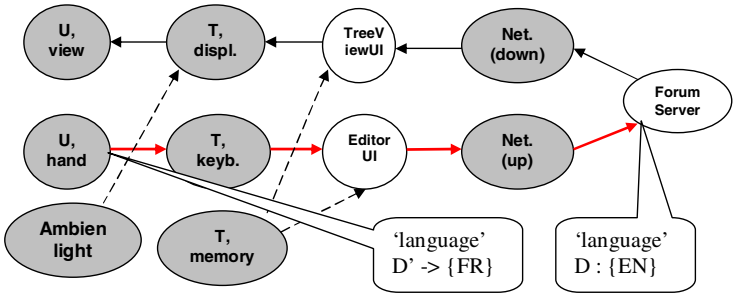

Fig. 6. Solution search for a flow related inadequacy

In principle, a new filter may be inserted at any point of a filter chain if two conditions are fulfilled: a) the IDL (Interface Definition Language) is compatible and b) it does not create a new inadequacy because of its own profile. An existent filter may be replaced with another one if the same conditions are fulfilled. As an observation, the service reconfiguration is not the only way to adapt the service-context system. In some cases, the system may suggest to user to do himself something that leads the service-context system into an adequate state, for instance, move with its terminal to a place with better network coverage.

3.4 How can we use and extend the meta-model?

The control part, figure 7, uses a common-defined, service-independent table containing the attribute definitions and the operators for composition and validation. This table must be created by a human operator and the idea is to have e unique definition for all services. Also, the profile attribute names must be respected by the component developers while describing the profiles.

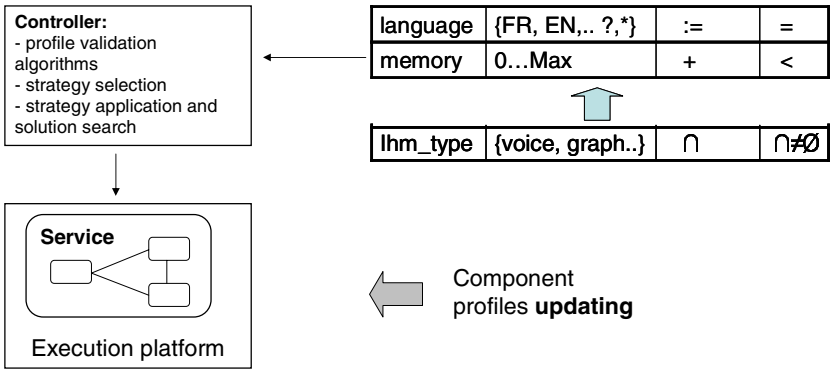

Fig. 7. Profile attributes definitions

Each component must have a profile, which means the component developer must describe that component as completely as possible. The component profiles

must be specified by the component developers and the adaptation depends on the profile content: if an attribute is not present in the profile we cannot check the adequacy for the missing attribute. The model extension requires additional lines in the attribute table, figure 7 and requires intervention of a human operator. Once added, these definitions may be reused. For the contextual components, common definitions must be used too. We may have a common ontology defining contextual components, for instance different terminal types with their characteristics.

3.5 Conceptual architecture for the autonomic adaptation platform

Figure 8 shows the proposed architecture that is based on the classical closed loop control principle.

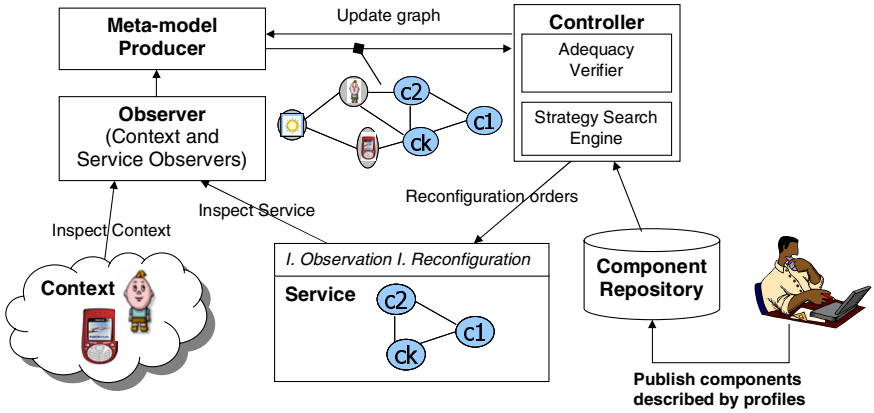

Fig. 8. Adaptation platform architecture

The platform is composed by the following components:

- The adapted *Service* is composed by several interconnected components and runs over an execution platform (i.e. a reflexive platform). This platform provides an observation interface (introspection) and a configuration interface offering reconfiguration primitives such as: create/destroy, bind/unbind, migrate, parameterise components and interconnections.
- The *Observer* module extracts information about the context and the service. The context and the service are observed using some dedicated components called *observers*. Context elements may be discovered dynamically, using pattern recognition on a webcam recorded image for instance. The service observers concern for instance the amount of memory or processor time used by a component.
- The *Meta-model Producer* produces and updates periodically the meta-model graph (see section 3.1). This module puts together all the information obtained about the service and the context.
- The *Controller* has two functions: a) to check the service to context adequacy (see section 3.2) by analysing the service-context graph and b) to search the right strategy if the service is not adequate. The service is reconfigured through the execution platform reconfiguration interface.

- The *Component Repository* contains reusable components published by different component providers, including observer components. The *Controller* may use these components in order to adapt the service.

4 Prototype

The prototype architecture is described in figure 9. The *Component Repository* con-

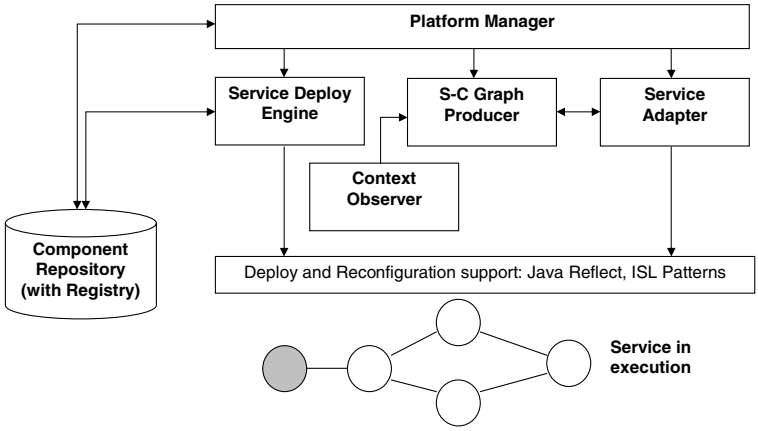

Fig. 9. Profile attributes definitions

tains the available components, in our case we have placed atomic components there, such as the three forum components (viewer, editor, server), a translation component, a language detector component and also the forum itself that is a composite component. Components are implemented in Java, we use CCM (Corba Component Model). Each atomic component has an IDL (Interface Definition Language) description and a profile described in XML. The forum is a composite component and its architecture is described using an XML based ADL (Architecture Definition Language). There is no need to specify profiles for composite components because their profiles are determined by the inner components' profiles (according to the graph model and the composition rules).

The *Service Deploy Engine* uses Java reflection API in order to deploy the service component instances and binds the components together. When a service is deployed, the *S-C Graph Producer* initializes the service and context graph. In this case, the context is implicit: the user is always connected to the service through the HMI and each service component always uses the host resources. This example takes into account only one contextual component: the user and only one information flow attribute: the language.

The *Context Observer* contains dedicated components also called 'observers' that offer the information needed in order to update the service-context graph. In this case, a graph analysis reveals the fact that we have a user and the 'language' attribute for this user is unknown. The solution is to search an observer component capable to detect the value of this attribute. We look in the *Component Repository* and we find a language detector component. Now, the question is how to collect

the necessary information in order to detect the user language? If we analyze the service-context graph (see fig. 6), we see that the informational flow passes from the user to the forum server. The language detector interface specification (ISL) does not allow us to insert it at the HMI level but we can do it at the *EditorUI* output. So, the language detector is inserted there. The value resulted at the language detector output will be used in order to update the service-context graph.

The *Service Adapter* re-checks the service adequacy to the context each time the service-context graph is updated. If the service is not adequate, a solution must be found. This prototype uses only one reconfiguration type, for the moment: component insertion. Thus, to find the solution means to find the right point where to insert the component (see section 3.3). Runtime reconfiguration is applied using interaction patterns described using the ISL (Interaction Specification Language) language [1]. ISL allows us to specify a general pattern that is then used to insert new components by interception. The interception means that a message/method call between two existent components may be intercepted and redirected towards a third component. We use this mechanism when we want to insert the translation component into the forum service architecture.

The *Platform Manager* implements the platform flow and assures the coherent communication between the other parts of the adaptive platform.

Figure 10 depicts the forum user interface. The terminal used in this case is a PDA. The user logs in, the platform detects a conflict between the user profile (if previously stored) and the service profile because the user language and the service language are different.

a) b) c)

Fig. 10. Forum UI for a PDA device

The platform proposes two possibilities to the user: either use a translator or leave the service unchanged. Assuming that the user chooses the translator from French to English, his messages are translated. The prototype has two versions: in the first one the user language is supposed to be stored in a database, in the second one the language is detected at each message. In the second version the user may

write a message in English, French and German and the service is reconfigured for each message, dynamically.

This prototype shows that the forum service may be adapted without using service-specific rules, without indicating the component that must be inserted and without specifying the insertion point (strategy). Instead, the language related inadequacy is discovered by analyzing the service-context model; the required component and the insertion point are searched for.

5 Evaluation, conclusions, perspectives

According to our approach, autonomic service adaptation requires to solve two major problems. The first one is to automatically determine the adequacy between a service and its context. The second one depends on the first one and it is to automatically find adaptation strategies leading the service-context system towards an adequate state. In this paper we have proposed a solution for the first problem, the most important one. This solution is based on a meta-model describing the service, its context and their interactions. The adequacy may be automatically determined using service and context-independent rules and operators.

Our prototype has shown that it is possible to adapt a simple service using the proposed meta-model, without using service and context specific rules/strategies. The strategy search problem was only partially solved: we have shown that it is possible to discover the place where a new component may be inserted but we have fixed the strategy type to component insertion. However, the proposed model enables the machine to generate different service-context configurations and check their adequacy. The system is proactive; it discovers new adaptation possibilities function on available components. If several solutions exist, the user may also decide which one to use.

Our proposal redistributes the developer effort for building adaptive services toward components developers. Instead of predicting the context and writing particular adaptation rules/strategies, each component developer must describe his components using profiles. A common attribute definition (ontology) must be used by all developers. One advantage of the profiles is reutilization: a profile of a component composed itself by several components having each one a profile can be computed automatically. This means that only atomic components must be described by 'hand'. A drawback is the profile updating: if the component developer wants to add something in a component's profile a posteriori, the update must be done for all the services that already use the component.

The following issues are considered for future development: increase the accuracy for expressing the service adequacy to the context, autonomic discovery of new contextual components, strategy selection and search algorithm improvement (using some AI inference engines), feedback-based learning.

References

[1] Blay-Fornarino, M., A. Charfi, D. Emsellem, A.-M. Pinna-Dery and M. Riveill, *Software interactions*, Journal of Object Technology **3** (2004), pp. 161–180.

[2] Brézillon, P., *Context-based modeling of operators' practices by contextual graphs*, in: R. Bisdorf, editor, *Proceedings of the Human Centered processes* (2003), pp. 129–137.

[3] Chan, A. and S.-N. Chuang, *Mobipads: a reflective middleware for context-aware mobile computing*, IEEE Transactions on Software Engineering **29** (2003), pp. 1072–1085.

[4] Chen, G. and D. Kotz, *Context-sensitive resource discovery*, in: *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, 2003, pp. 243–252.

[5] Dey, A., *Understanding and using context*, Personal and Ubiquitous Computing **5** (2001), pp. 4–7.

[6] Dowling, J. and V. Cahill, *The k-component architecture meta-model for self-adaptive software*, in: *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns* (2001), pp. 81–88.

[7] Dubus, J. and P. Merle, *Vers l'auto-adaptabilité des architectures logicielles dans les environnements ouverts distribués.*, in: *CAL*, 2006, pp. 13–29.

[8] Floch, J., S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund and E. Gjorven, *Using architecture models for runtime adaptability*, IEEE Software **23** (2006), pp. 62–70.

[9] Fujii, K. and T. Suda, *Semantics-based dynamic service composition*, IEEE Journal on Selected Areas in Communications **23** (2005), pp. 2361– 2372.

[10] Garlan, D., S.-W. Cheng, A.-C. Huang, B. Schmerl and P. Steenkiste, *Rainbow: Architecture-based self-adaptation with reusable infrastructure*, Computer **37** (2004), pp. 46–54.

[11] J., K., "Completely Unanticipated Dynamic Adaptation of Software," Ph.D. thesis, University of Dublin, Trinity College, Distributed Systems Group (2004).

[12] Jacob, B., R. Lanyon-Hogg, D. K. Nadgir and A. F. Yassin, "A Practical Guide to the IBM Autonomic Computing Toolkit," IBM (2004).

[13] Kephart, J. O., *Research challenges of autonomic computing*, in: *ICSE'05: Proceedings of the 27th international conference on Software engineering*, 2005, pp. 15–22.

[14] Roman, M., "An Application Framework for Active Space Applications," Ph.D. thesis, University of Illinois at Urbana-Champaign (2003).

[15] Wang, X. H., D. Q. Zhang, T. Gu and H. K. Pung, *Ontology based context modeling and reasoning using owl*, in: *PERCOMW '04: Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops* (2004), p. 18.