A Self-Adaptation of Software Component Structures in Ubiquitous Environments

Gautier Bastide
Ecole des Mines de Douai
941 rue Charles Bourseul
59508 Douai, France
bastide@ensm-douai.fr

Abdelhak Seriai
Ecole des Mines de Douai
941 rue Charles Bourseul
59508 Douai, France
seriai@ensm-douai.fr

Mourad Oussalah
LINA, Université de Nantes
2 rue de la Houssinière
44322 Nantes, France
oussalah@univ-nantes.fr

ABSTRACT

The creation of applications able to be executed in ubiquitous environments, involves a better consideration of the execution context in order to ensure service continuity. In component-based software engineering, applications are built by assembling existing components. For deploying such applications in ubiquitous environments, its components must be able to adapt themselves to the current context. To deal with this issue, we propose in this paper an approach aiming at reconfiguring the component structure to allow a flexible deployment of its services according to its use context. This adaptation focusing on the service continuity, consists of determining a structure adapted to the execution context. Then, the current structure is automatically reconfigured and the generated components are redeployed.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Algorithms, Design, Measurement, Performance

Keywords

Software component, self-adaptation, restructuring, deployment, ubiquitous systems, context-awareness, clustering

1. INTRODUCTION

For several years, ubiquitous computing has emerged as a challenge field for application design. In fact, due to the huge development of mobile devices, designers have to create applications able to adapt themselves to the new conditions which can modify service continuity. For example, a user device can have scarce resources, such as low battery power, slow CPU speed and limited memory. So when an application is executed on such a device, these hardware resources can become insufficient for guaranteeing service continuity. In this case, the application must adapt itself.

In addition, more and more applications are created by assembling reusable parts. In Component-Based Software Engineering (CBSE), an application consists of existing components [4]. To be executed in ubiquitous environments where context is perpetually evolving, a component must adapt itself. Adaptation can concern component behavior or component structure. Existing work [2, 3] focuses on adapting component behavior and few works are related to the structure. Besides monolithic components are considered as deployment units which cannot be structurally adapted. In fact, the existing work focuses on the placement of components or sub-components within a distributed infrastructure.

However, adapting the component structure can be required in many cases. To illustrate this, let us consider an example of a monolithic component which has to be deployed on an infrastructure composed only by resource-constrained machines. Moreover, no machine can deploy the component because their resources are insufficient. So the component has to be adapted. This adaptation can be achieved by fragmenting the component structure and by distributing some of its services through the infrastructure. Now, imagine that a sudden fall in the available memory involves a breakdown in its service continuity. So the component must adapt itself to these conditions since runtime.

While being based on the above considerations, we propose an approach aiming at reconfiguring component structure in order to achieve a flexible deployment of its services.

In previous work [1], we developed an approach allowing an application administrator to adapt the software component structure since runtime. However, due to the continual context evolution, a manual decision-making cannot be achieved. So we propose to automate this task.

We discuss our approach as follows. Section 2 presents the self-adaptation of component structures. Section 3 details the decision-making mechanisms. Conclusion and future work are presented in section 4.

2. STRUCTURAL SELF-ADAPTATION

2.1 Definitions and motivations

In ubiquitous environments, the context is always changing. So a component which has been deployed on a resource-constrained device, is not able to guarantee its service continuity without adapting itself to the current hardware architecture. This adaptation can consist in fragmenting and distributing its structure according to the available resources.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICPS'08, July 6–10, 2008, Sorrento, Italy.
Copyright 2008 ACM 978-1-60558-135-4/08/07 ...$5.00.

For example, if a component is not able to guarantee the continuity of all its provided services, some of its services may be distributed on the available infrastructure.

Figure 1 shows a such component (C_1) which is deployed on a resource-constrained machine called *host1*. Let us consider that, during the component runtime, existing load balancing has become unfit because of the deployment infrastructure evolution. In this case, C_1 must adapt itself in order to ensure service continuity. A solution can consist of the component fragmentation into four new components called C_2, C_3, C_4, C_5 providing each a subset of services provided by the component C_1. Then, the new components are deployed on the available infrastructure: C_2 is deployed on *host1*, C_3 and C_4 on *host2* and finally C_5 on *host3*. Thus, the component C_1 preserves its service continuity on *host1* although its sub-components are distributed.

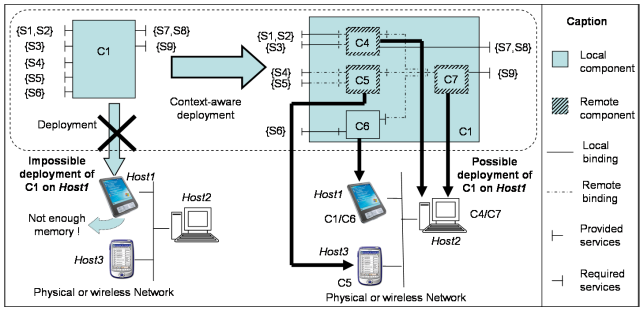

Figure 1: Adaptation of a component structure

We define the self-adaptation of a component structure as the property of a component to update its structure itself according to its current context. Thus, a component having such a property must be able to acquire data related to its context in order to generate an adapted structure. In fact, a self-adaptation process is composed of three steps:

(1) context acquisition: a self-adaptive component has to acquire data related to its execution context. However, all contextual data have not an impact on the component adaptation. So, we need to determine the context elements which can affect a component structure.

(2) decision-making: the contextual data are interpreted in order to release an adaptation process and to generate a specification of an adapted structure.

(3) adaptation achievement: the last step of an adaptation process consists in updating the component structure in order to make it conform to the generated specification and redeploying it if it is necessary.

2.2 Structure-dependent context

First of all, we have to define the context elements which can affect a component structure. These elements will be used in order to determine when an adaptation process has to be release and how a structure specification adapted to the current context is generated. We distinguish three kinds of context on which a component structure depends.

Hardware architectural context: first, the component structure depends on the infrastructure technical features. In fact, a component service can be deployed on a site only when the resources required by this service are provided by this site. In addition, as some of these resources are perpetually evolving, they can become insufficient to ensure the continuity of the services deployed on a site. So these data must be used to release an adaptation process.

Environmental context: second, the component structure depends on the environmental context. In fact, the services that show the highest probability to be used have to be deployed on the user device or on its neighborhood when the required technical features are sufficient to ensure service continuity. The use probability must be determined by comparing the similarities between the user-target profile and the current user profile, and between the foreseen use-conditions and the current conditions. Neighboring nodes are determined according to infrastructure features (e.g. best bandwidth). This strategy can be useful in ubiquitous environments because of the connection volatility and variations of network performance. So several services can be used in spite of the disconnections of the user device. Furthermore, the fixed devices must be privileged in order to limit the risks of disconnections related to the other deployment sites of the component. The environmental context contains two kinds of contextual data: user profile (e.g. private data) and use conditions (e.g. user activity).

Software context: finally, the component structure depends on the component behavior. In order to optimize component execution (by minimizing the number of remote connections) and to minimize the consequences of disconnections (by guaranteeing the completeness of services deployed on the same host), the services which are closely dependent should be deployed on the same site. The software context contains behavioral data which are acquired through an application history used to memorize events occurred in the past (e.g. service call, adaptation-release context).

2.3 Self-adaptive component model

To achieve the structural self-adaptation, a component must be both self-adaptive and structural-adaptive. In order to be structural-adaptive, a component must conform to a canonical model, defined in [1], which allows it to update its structure for runtime. Such a component is a composite component whose each provided interface is reified into a primitive sub-component called interface-component. In order to be self-adaptive, a component must contain the three following components:

Context-manager: this component is used for the modeling of service use-cases and the management of contextual data (acquisition, interpretation and aggregation). Each interface-component is assigned to a description which contains two kinds of data: the list of required resources needed for the component deployment, and, data related to the user-target profile and the use conditions. Contextual data are acquired using sensors.

Decision engine: this component provides decision-making mechanisms allowing the component to specify automatically the adaptation script used by the structural adaptation process for generating the adapted component. This script is a description of the component structure, formulated through an ADL. It contains a description of the components to generate: for each component, its services and its deployment site are specified.

Adaptation engine: once the script defining the new structure has been generated, structural adaptation has to be achieved by the component using the reconfiguration engine then the deployer engine. The functionalities of these two engines are described in [1].

3. DECISION-MAKING FOR THE GENERATION OF AN ADAPTED STRUCTURE

3.1 Strategy overview

Contextual data are used by the component to generate a specification corresponding to a new component structure which matches with the current context: each service provided by the component to adapt must be associated with a deployment site. Each set of services associated with a site are merged into a same sub-component. To generate such specification of the new component structure, we identify three kinds of classification tasks which differ according to their impact on the component structure.

First, the environmental context is used for classifying services provided by the adapted component according to their priority of deployment on the user device. In fact, this priority depends on similar points between the context target (defined by the application administrator) and the current context (acquired using sensors or user data). However, these classification tasks cannot be generated automatically because events and decisions are specific to the application. In this case, the selection strategy must be designed by the application administrator using rules like ($< condition > \Rightarrow < action >$).

The second task is based on selecting sites where services are able to be deployed, taking into account the resources available on the different nodes of the distributed infrastructure. This task is achieved by matching the service requirements (defined by the component designer) with the contextual data. We aim at generating a configuration which maximizes the number of high-priority services deployed on the user device or on its neighborhood. So, each subset of services associated with a site corresponds to a component generated during the structural adaptation. It will be redeployed on the corresponding site. However, this selection task cannot be entirely automated because of the specificity of the resources required by each service. In fact, this selection is achieved using adaptation policies defined by the component designer.

The last task is based on classifying the component services according to the data related to the component structure and behavior. This classification is used to optimize the service distribution by evaluating the dependences between services. In fact, the optimization is based on minimizing remote connections between the new specified components by merging the most dependent services within sub-components which are deployed on sites according to their dependences with other application components. Contrary to the two previous tasks, where a treatment specific to the application is required, this task can be entirely automated. This operation is detailed below.

3.2 Service-dependency awareness

The objective of software-awareness is to merge the most dependent services within sub components deployed only on one device. Dependences between services [1] are of two kinds: functional dependences and dependences related to the resource-sharing. They require the introduction of remote communications between components (e.g. remote service call, shared-resource synchronization); whose cost can be substantial. So, sub-components must be specified taking into account these dependences in order to minimize the remote communications.

3.2.1 Evaluable context-elements

To set up such selection mechanisms, we need to quantitatively evaluate dependencies among components. To do so, we have to create a history of communications among sub-components (software context). Data collected for each service (S_i) of the self-adaptive component are the following:

(a) the probability that the service S_i calls S_j with $S_j \in S_{provided} \cup S_{required}$, noted $P_{use}(S_i, S_j)$. This probability is related to the number of S_j call during S_i execution (direct or indirect call) compared to the service S_i call number,

(b) the average number of calls from S_i to S_j with $S_j \in S_{provided} \cup S_{required}$, noted $M_{aver}(S_i, S_j)$,

(c) the average number of parameters used when a service S_j is called by S_i, noted $Nb_{param}(S_i, S_j)$ as well as the average memory size (in bytes) of these parameters, noted $T_{param}(S_i, S_j)$,

(d) the probability to update a resource, in S_i, shared with S_j such as $S_j \in S_{provided}$, noted $P_{update}(S_i, S_j)$,

(e) the probability to initiate a critical section in S_i related to a resource shared with S_j such as $S_j \in S_{provided}$, noted $P_{critical}(S_i, S_j)$.

3.2.2 Service-dependency evaluation

The software-context elements are used to evaluate proximities between the various services provided by the adapted component. In fact, the proximity between two services depends on their coupling (*i.e.* evaluation of the functional dependences) and on their cohesion (*i.e.* evaluation of the dependences related to resource-sharing).

The coupling (1) between two different services S_i and S_j noted $C_{coupling}(S_i, S_j)$ is evaluated according to the probable number of calls of the service S_j since the execution of S_i and inversely (2) balanced by the number and the type of parameters exchanged between these two services (3). In fact, this weight is computed according to the average number of parameters used for the service call and the average memory size of these parameters. Thus, we evaluate the coupling between two services S_i and S_j as follow:

$$C_{coupling}(S_i, S_j) = \alpha(S_i, S_j) * \beta(S_i, S_j) + \alpha(S_j, S_i) * \beta(S_i, S_j) \quad (1)$$

$$Where \quad \alpha(x, y) = T_{param}(x, y) * (Nb_{param}(x, y) + 1) \quad (2)$$

$$\beta(x, y) = M_{aver}(x, y) * P_{use}(x, y) \quad (3)$$

The cohesion (4) between two different services S_i and S_j, noted $C_{cohesion}(S_i, S_j)$ is evaluated according to the number of critical sections started in each service (5) and the frequency update of resources shared between S_i and S_j and inversely (6), balanced by the number and the type of resources (7). The weight related to the type of resources corresponds to their average memory size expressed in byte and noted $T_{sr}(S_i, S_j)$. Thus, the value of cohesion between two services S_i and S_j are obtained as follows:

$$C_{cohesion}(S_i, S_j) = \gamma(S_i, S_j) * \eta(S_i, S_j) + \chi(S_i, S_j) \quad (4)$$

$$Where \quad \chi(x, y) = P_{critical}(x, y) + P_{critical}(y, x) \quad (5)$$

$$\eta(x, y) = P_{update}(x, y) + P_{update}(y, x) \quad (6)$$

$$\gamma(x, y) = Nb_{sr}(x, y) * T_{sr}(x, y) \quad (7)$$

The proximity between two services S_i and S_j contained in a set S is a binary relationship, noted $Pr(S_i, S_j)$ defined as follow:

$$Pr(S_i, S_j) = \begin{cases} 1 & \text{if } S_i = S_j \\ \frac{(\alpha * C'_{coupling}(S_i, S_j) + \beta * C'_{cohesion}(S_i, S_j))}{\alpha + \beta} & \text{else} \end{cases}$$

Where

$$C'_{coupling}(S_i,S_j) = \begin{cases} 0 & \text{if } C_{coupling\_max} = 0 \\ \frac{C_{coupling}(S_i,S_j)}{C_{coupling\_max}} & \text{else} \end{cases}$$

$$C'_{cohesion}(S_i,S_j) = \begin{cases} 0 & \text{if } C_{cohesion\_max} = 0 \\ \frac{C_{cohesion}(S_i,S_j)}{C_{cohesion\_max}} & \text{else} \end{cases}$$

$$C_{coupling\_max} = max(\{C_{coupling}(x,y), \quad \forall x,y \in S \quad / \quad x \neq y\})$$

$$C_{cohesion\_max} = max(\{C_{cohesion}(x,y), \quad \forall x,y \in S \quad / \quad x \neq y\})$$

The proximity between two services varies from zero, when the two services are not dependent, to one. The value one means that the two services are identical (by convention). The closer to one the value of the proximity is, the more dependent the two services are. α and β are the impact factors for respectively coupling and cohesion. According to the use needs, the application administrator can instantiated these factors by giving more weight to the coupling ($\alpha > \beta$) or to cohesion ($\alpha < \beta$). By default, we supposes that these two dependences have identical impacts ($\alpha = 1$ and $\beta = 1$).

3.2.3 Clustering-service algorithm

The evaluation of proximities between the services provided by the adapted component is used to merge services in subsets containing the most dependent services, among themselves. Each subset constitutes a component whose the provided services are the ones contained in this subset.

In addition, the dependencies between provided and required services can be used to determine the deployment site of each generated component. The goal is to minimize remote communications with the other application components providing the services required by the component to adapt. In fact, the generated components must be deployed on the sites which provide services close to the services provided by the adapted component.

To obtain an interface partition whose elements are associated with a deployment site, we use a hierarchical clustering algorithm (Fig. 2). It requires as a parameter, an array whose cells contain an evaluation of proximities between provided services and with deployment node of the infrastructure. So we can note that there are two kinds of clusters: on the one hand, service-clusters which contain a set of services provided by the adapted component, and, on the other hand, site-clusters which contain the set of services provided by components which are deployed on it and which are required by the adapted component.

$Cl_{init} \leftarrow \{Set \quad of \quad initial \quad clusters\}$
While $\exists Cl_i \in Cl_{init}$ such as $Cl_i \cap Sites \neq \oslash$ Do
  $\forall Cl_i \in Cl_{init}, \forall Cl_j \in Cl_{init} \cup Sites, T[Cl_i, Cl_j] \leftarrow Pr(Cl_i, Cl_j)$
  Find Cl_{maxi} and Cl_{maxj} such as
    $\forall Cl_i \in Cl_{init}, \forall Cl_j \in Cl_{init} \cup Sites, \quad Cl_i \neq Cl_j$ and
      $T(Cl_{maxi}, Cl_{maxj}) \geq T(Cl_i, Cl_j)$ and $|Cl_{maxi} \cap Site| \leq 1$
  $Cl_{init} = Cl_{init} \cup \{(Cl_{maxi}, Cl_{maxj})\}$
  If $Cl_{maxj} \in Cl_{init}$ then
    $Cl_{init} = Cl_{init} - \{Cl_{maxi}, Cl_{maxj}\}$
  If $Cl_{maxj} \in Sites$ then
    $Cl_{init} = Cl_{init} - \{Cl_{maxi}\}$
    $Sites = Sites - \{Cl_{maxj}\}$
Return Cl_{init}

Figure 2: Service-clustering algorithm

The main idea behind this algorithm is to merge the services provided by the adapted component, in clusters, according to their proximity. And, each cluster must be associated with only one deployment site. Initially, the maximal value of the array is searched. Two cases can appear: if this value corresponds to the proximity between two service-clusters, these two clusters are merged. If this value corresponds to the proximity between a service-cluster and a site-cluster then the service-cluster is associated with the corresponding site. If the cluster is already associated with a site, this value is ignored and the algorithm searches the maximal value of this array except this cell. Once a fusion or an association has been achieved, the proximity array is evaluated again according to the new clusters. Two solutions are possible to calculate the proximities between two clusters: either all proximity values (coupling and cohesion) are evaluated again according to the services contained in each cluster, or an approximation of the proximity between two clusters is done. The former cannot be considered because of its complexity ($O(n^6)$) which cannot be acceptable for runtime adaptation. That is why, we chose a lower-complexity strategy ($O(n^2)$) based on the average proximities between the cluster elements:

$$Pr(C_{l_1}, C_{l_2}) = \frac{1}{|C_{l_1}||C_{l_2}|} \sum_{S_i \in C_{l_1}, S_j \in C_{l_2}} Pr(S_i, S_j)$$

These operations are reiterated until each cluster is associated with a deployment site. Then, the result corresponds to the different clusters obtained and their associated site.

4. CONCLUSION AND FUTURE WORK

We presented an approach aiming at the component structure reconfiguration in order to allow a flexible deployment of its services. Such components must conform to a canonical format which is based on interface reification. Besides, it must integrate mechanisms enabling it to acquire and analyze its context, to determine an adapted structure guaranteeing its service continuity. Then, the specified structure is generated by encapsulation of interface-components within new sub-components which can be redeployed independently.

As mentioned in our motivations, our approach aims at allowing a flexible deployment of software components in order to ensure their service continuity. However, our adaptation process involves an overhead related to the management of the communication and synchronization between the generated sub-components and decision mechanisms. Currently, we evaluate this overhead.

5. REFERENCES

[1] G. Bastide, A.-D. Seriai, and M. Oussalah. Software component re-engineering for their runtime structural adaptation. In Proc. of the Int. Conf. on Computer Software and Applications (COMPSAC), pp. 109-114, 2007.

[2] P. Boinot, R. Marlet, J. Noye, G. Muller, and C. Consel. A declarative approach for designing and developing adaptive components. In Proc. of the Int. Conf. on Automated Software Engineering (ASE), p. 111, 2000.

[3] J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In Proc. of the Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns, pp. 81-88, 2001.

[4] C. Szyperski. Component software: beyond object-oriented programming. ACM Press, 1998.