An Adaptation Methodology for the Deployment of Mobile Component-based Applications

Dhouha Ayed*, Chantal Taconet†, Guy Bernard‡ and Yolande Berbers*

*Department of Computer Science, K.U.Leuven
B-3001 Leuven, Belgium
Email: {dhouha.ayed,yolande.berbers}@cs.kuleuven.be

†GET / INT, CNRS Samovar 5157
9 rue Charles Fourier, 91011 Évry, France
Email: {Chantal.Taconet, Guy.Bernard}@int-evry.fr

Abstract—The expansion of wireless communication and mobile hand-held devices is affecting how software deployment is being performed. Deployed applications have to be suited to the users requirements, to the resources of his terminal and to the surrounding environment. In this paper, we define a methodology which enables classical deployment tools provided by component-based middlewares to be adaptive. This methodology was experimentally tested and evaluated.

I. INTRODUCTION

Deployment refers to all activities performed after the development of the software which make it available to its users. These activities consist essentially in installing and configuring the software but can also include software reconfiguration, updates and even un-installation.

With the continuous growth of wireless communication, mobile hand-held devices such as PDAs and mobile phones are becoming powerful platforms that allow users to utilize a whole range of entertainment and business applications.

Hand-held devices have scarce resources, such as low battery power, slow CPU speed and limited memory. They rarely have a secondary memory.

The specificity of mobile users is that they undergo constant changes in their context, such as a high variability of network bandwidth, location and physical environment. A mobile user needs to be able to deploy the same application on different execution environments and in different contexts. For example, he must be able to deploy the same application on his PDA in his car, on his PC at his office or on a friend’s laptop when he is not at home. These varied terminals have different hardware and software environments. Hence, the deployment needs to be processed differently according to the environment and the user’s situation.

The resource limitation of hand-held devices and the high variability of contexts lead mobile users to perform several repetitive deployment activities, which they do for the following reasons: (1) The user has to install his applications on all his different types of terminals. (2) If the user’s terminal does not have a secondary memory, then the applications have to be redeployed each time they are required. (3) If the user’s terminal has a secondary memory, this memory generally has a limited size that does not allow the installation of all the applications needed by the user. Thus, he might have to uninstall some applications in order to install others. When he needs the first applications again, then he has to reinstall them. (4) When the context changes, the user has to reconfigure the installed applications according to the context.

For all these reasons, mobile environments require the deployment process to be context-aware and entirely automated. Context-awareness will satisfy the user’s requirements, which are constantly changing according to the context, and automation will relieve the user of repetitive deployment tasks.

The purpose of this paper is to propose a methodology that adapts the deployment of component-based applications for mobile users. Section 2 briefly introduces the requirements and issues of the software deployment tools used in mobile environments. Section 3 gives an application example. The deployment adaptation methodology is introduced in Section 4. Section 5 specifies the deployment adaptation rules used in this methodology and Section 6 discusses their consistency. Section 7 presents the adaptive deployment algorithms. Section 8 discusses the implementation status and the experimental results and Section 9 discusses some related work before the paper concludes in Section 10.

II. REQUIREMENTS OF DEPLOYMENT TOOLS USED IN MOBILE ENVIRONMENTS

The deployment issues in mobile environments mentioned in the introduction lead to the requirements described below:

• Deployment Automation: To relieve users of the repetitive deployment tasks, the installation, configuration and activation of the different applications must be carried out automatically by the deployment tool. The user just has to choose the application he wants to deploy and the remaining tasks will be carried out transparently without any user intervention.

• Adaptive Deployment: As the user undergoes frequent context changes, the deployment tool has to be context-aware. It must take into account the resources of the user’s terminal, the user’s activity, his preferences and his physical environment, in order to deploy the required applications. Context-aware deployment must satisfy the
user’s requirements, which change according to the context, and it must optimize the consumption of resources.

- **Just-in-time Deployment**: The applications must be deployed only when accessed by the user and undeployed automatically just after their deactivation. Deployment of several applications on the user’s terminal reduces the available memory on the terminal and slows down the execution of the newly deployed applications. Thus, just-in-time deployment contributes to saving terminal resources and it enables a better adaptive deployment.

- **Deployment Without Delays**: Deployment adaptation and automation should not add significant delays compared to classical deployment. The delays added by the adaptation should not be perceptible by the user.

III. **An Example of a Possible Application**

To illustrate our approach, we give the example of an online shopping application for mobile users.

The idea consists in deploying an application for mobile users which allows them to virtually visit the nearest store departments and order the goods they want to buy before arriving at the store. When the customer arrives at the store, he has only to recover the ordered goods. This procedure, allows the customer to save time.

The application consists of four components: A Graphical User Interface (GUI) component, a Data Base (DB) component, a Local View (LV) component and a Cart component.

The DB component is hosted in a proximity server and contains the information about the goods i.e. their characteristics and prices. The GUI allows the user to visit the store departments, to choose the goods he wants to buy and to put them in the Cart component which is deployed on the user terminal. The GUI component has two implementation versions. A normal screen version which is dedicated to desktop and laptop users and a little screen version which is dedicated to PDA users.

If the user chooses to work in a disconnected mode or he is in a zone near to a non-covered zone, he may choose one or several departments of the online store whose information will be placed in an LV component. The LV component is deployed only in these two cases and has normally to be placed on the user terminal. The LV presents a view of the DB component which contains only information related to the selected departments.

When the user terminal has limited resources and that he is not situated near to a non-covered zone the Cart component and the LV component can be instantiated in a proximity server.

Each time the user asks to deploy the application, he will be connected to the DB instance of the nearest store. Figure 1 shows two possible architectures of this application and Figure 2 shows two examples of possible placements of the Cart component and two possible implementation versions of the GUI component.

The components of the online shopping application can be used with several languages.
The deployment of component-based applications in distributed systems requires the description of an application deployment plan. The deployment plan is meta-data that describes the following four deployment parameters of the application (cf. Figure 3):

- **(A) The application architecture.** This parameter describes the components making up the application and the connections between them.
- **(B) The placement of each component** on the nodes of the deployment domain.
- **(C) The component implementation version**
- **(D) The property values** that the components will be instantiated with.

The deployment plan is used by the deployment tool during the deployment process to instantiate the application components where they are expected to be placed and to configure them by using the information supplied about the property setting. Once all of the components are instantiated, they are connected.

This deployment process, which is used by several deployment services provided by component-based middleware [1] [2] [3] is static. The structure of the application is fixed for any context and the placement of the components must either be defined before the deployment of the application without studying the node capacity or else it must be fixed by the user manually at the deployment time.

To adapt the four deployment parameters to the context, we propose adapting the four different deployment parameters that we identified according to the context. For example, the architecture of the “online shopping” application depends on the network connection and the user preference which determine whether the LV component and its connections to the GUI and DB components will be deployed or not. The other examples of deployment parameters are the choice of the implementation version of the GUI component according to the screen size of the user terminal, the placement choice of the Cart component according to the resources availability, and the variability of the language property of the different components according to the used language.

To adapt the four deployment parameters to the context, we propose to dynamically generate the deployment plan according to the context. For this purpose, we define and develop a set of adaptation rules and adaptation algorithms. These algorithms process the rules for dynamically creating a deployment plan after studying the current context state (cf. Figure 4).

![Fig. 3. The Deployment Plan Structure](image)

The adaptation rules are discussed in Sections 5 and 6 and the adaptation algorithms are presented in Section 7. The implementation of the framework supporting this methodology is presented in Section 8.

V. DEPLOYMENT ADAPTATION RULES

We define an adaptation rule of a deployment parameter (defined in Section IV) as an association between a context constraint and a value of this parameter. It can also be defined as an association between a context constraint and a context mapping which maps the value of the parameter to a given context. It is generally written as follows:

\[\text{constraint} \implies \text{parameter} = \text{value}, \text{ or } \text{constraint} \implies \text{parameter} = \text{ContextMapping}(\text{ctx})\]

Each deployment parameter is associated with a set of adaptation rules that specify the different values that it can take.

A context constraint represents a particular state of the context. For example, the “TerminalType” context can take different states such as “TerminalType=PDA”. A context constraint can be a composition (conjunction and/or disjunction) of several context states such as “userLocation=zoneA \land userLanguage=Arabic”.

As an application is an assembly of components, we define two levels of rules: The component level rules and the application level rules:

- **The component level rules** are the adaptation rules that are specific to a given component. The component developer who does not know in which application the component will be deployed, specifies these types of rules. This is why these rules take into account the component semantics without considering the application semantics in which they will be deployed.
- **The application level rules** are the adaptation rules that are specific to an application to be deployed. The application assembler who takes into account the application semantics specifies them.

In the following we detail each type of rule. Normally, we specify the adaptation rules in XML but the examples that we will give in this paper will be specified in a simple manner (without any specific language) in order to be more readable.
A. Component Level Rules

The component developer can specify the adaptation rules of two deployment parameters at the component level: the adaptation of the properties according to the context (parameter D) and the choice of the implementation versions (parameter C).

1) Properties Adaptation Rules: A component can have several functional configuration properties. For each of these properties, the component developer has to specify a default value, and the values that the property can take on, according to the context. Figure 5 shows the properties adaptation rules of the LV component. This component has two properties: the first property represents the size of the space where the component will put the the view of the DB component and the second one represents the user language supported by the component. The value of the first property varies according to the size of the available memory, and the value of the second property is determined by making a mapping on the context that represents the user language.

![Fig. 5. Properties of the LV Component](image)

2) Rules of Choice of Implementation Versions: A component can have several implementation versions dedicated to various execution contexts. The developer has to specify the context constraints required by each of these versions.

An implementation version can have two types of constraints: strong constraints and weak constraints. An implementation version cannot be selected during deployment if its strong constraints are not satisfied. Satisfaction of low constraints is not obligatory though it is preferable.

Figure 6 shows an example of a constraints description of the various implementations of the Cart component of the "online shopping" application.

![Fig. 6. Implementations of the Cart Component](image)

These constraints are used in rules in this way :

\[\text{constraint}_1 \land \text{constraint}_2 \land \ldots \land \text{constraint}_n \implies \text{ComponentImpl} = \text{implementation}_i\]

ComponentImpl = implementation_1  \\
\text{constraint}_j \land \text{constraint}_k \land \ldots \land \text{constraint}_z \implies \text{ComponentImpl} = \text{implementation}_n

A placement algorithm processes these rules (cf. Section VII). It checks whether the context satisfies the implementations constraints and it chooses the implementation suited to the context.

B. Application Level Rules

These rules specify essentially the adaptation of the application architecture according to the context (parameter A). But they can also complete the component level rules that depend on the application semantics. They do that by specifying the placement site (parameter B) and the configuration of the properties (parameter D) of the components making up the application.

1) Architecture Adaptation Rules: When the application assembler writes the architecture adaptation rules of the application, he distinguishes between two types of components: the obligatory components, which are deployed whatever the context state, and the optional components, whose existence in the final deployment plan depends on the context. In the online shopping application, the GUI component, the Cart component and the DB component are obligatory components. The LV component is an optional component since its existence depends on the context.

The application assembler specifies for each component of the application if it is obligatory or optional and for each optional component he specifies a condition which determines its existence. This condition represents a disjunction of several context constraints. For example, the existence condition of the logging component is described in this way:

\[(\text{userzone} = \text{"nearnoncoveredzone"}) \lor (\text{loggingMode} = \text{"true"})\]

This type of condition is used in a rule in this way:

\[\text{ExistenceCondition} \implies \text{ComponentExistence} = \text{true}\]

The application assembler also distinguishes between two types of connections: the obligatory connections, which are deployed whatever the context state, and the optional connections, whose existence in the final deployment plan depends on the context. Optional components also require the specification of a condition that determines their existence. This condition has the same structure as the optional component.

The establishment of an optional connection is possible only if the existence condition of the connection is satisfied and the component that it will connect to is deployed i.e. the components that it will connect to are either obligatory, or else they are optional but their existence condition is satisfied. For example, the optional connection between the DB component and the LV component can be established only if the existence condition of the LV component is satisfied.

The architecture adaptation rules require a consistency check which is presented in Section VI-B.

2) Complement of the component level rules: Besides the architecture of the application, the adaptation rules of the application level can specify the placement site and the properties adaptation of the components making up the application.
The rules described at this level are prior to those described at the component level, since they take into account the semantics of the application.

The description of the properties adaptation rules at the application level is specified in the same manner as the component level.

The placement of a component is performed by the specification of one or several logical node names associated with a context constraint as follows: $constraint_i \rightarrow ComponentDestination = logicalNode_i$ $(i = 1..n, n$ is the number of possible placements)$

A logical node represents a description of a set of node properties. A physical node corresponding to the logical node is determined at the deployment time through a discovery service [4].

The placement specification of a component complete the rules of choice of the implementation versions of the component level. If no placement rule is specified for a given component at the application level, the placement algorithm (described in Section VII) will automatically perform the component placement by using the rules of choice of implementation versions at the component level.

The LV component is an example of a component whose placement must be specified at the application level. If no placement rule is specified for the LV component, the placement algorithm will automatically choose any node that satisfies the constraints of one of the implementations of the logging component described at the component level. However, if we consider the semantics of the "online shopping" application, we know that the LV component must be placed on the user terminal if the user deploys the application when he is in a zone close to a non-covered zone. The placement of the LV component in a node other than the user terminal does not make sense in this context state since the user will be disconnected from the other sites. Consequently, the placement of the LV component must be specified at the application level, as in Figure 7.

A logical node represents a description of a set of node properties. A physical node corresponding to the logical node is determined at the deployment time through a discovery service [4].

VI. CONSISTENCY CHECK OF THE DEPLOYMENT ADAPTATION RULES

A logical node represents a description of a set of node properties. A physical node corresponding to the logical node is determined at the deployment time through a discovery service [4].

We have two types of inconsistency:

- An inconsistency in the description of the rules constraints. This type of inconsistency can occur at the application or component level and causes a contradiction between the rules associated with the same deployment parameter.
- An inconsistency in the description of the application architecture. This type of inconsistency occurs only at the application level and causes the deployment of one or several detached components or connections. A detached component is a deployed component without the connections that connect it to the remaining components of the application. A detached connection is a deployed connection without the components it connects.

A. CONSISTENCY OF THE RULES CONSTRAINTS

We have an inconsistency, if in a given context state, the deployment rules assign more than one value to a deployment parameter. They can, for example, assign more than one placement site or more than one implementation version to a component instance, or give more than one possible value to one of its properties.

The contradiction between rules has two main causes:

- The rules’ constraints associated with the same deployment parameter are related to the same context types and there is an intersection between them.
- The rules’ constraints associated with the same deployment parameter are related to different context types but can be satisfied simultaneously.

1) Interesection Between Rules Constraints: Let us consider the two following rules:

- disk size > 50MB => implementationversion_k
- disk size > 80MB => implementationversion_l

If the available disk size is greater than 80MB, the two rules can be applied and each of them gives a different result. The first rule recommends the use of the implementation k and the second one recommends the implementation l.

The checking tool announces the intersections between the rules constraints to the component developer or to the assembler of the application according to the rule’s level and suggests a solution to him that the user can either accept or suggest another one.

2) Simultaneous Satisfaction of Rules Constraints: If there is no intersection between the constraints that are related to the same type of context, they can never be satisfied simultaneously because the context takes only one state at a given time. But if these constraints are related to different types of context, they can be satisfied in the same time. Let us consider this example:

Placement of the LV Component:

constraint_1: user zone = "near non covered zone" AND user terminal CPU use < 80% => user terminal
constraint_2: disconnectedMode = "true" AND user terminal CPU use > 80% => the nearest node to the user
constraint_3: disconnectedMode = "true" AND user terminal CPU use < 80% => user terminal

Fig. 7. Placement of the LV Component

- user zone = "near non covered zone" AND user terminal CPU use < 80%
- disconnectedMode = "true" => ComponentDestination = UserTerminal
- disconnectedMode = "true" AND user terminal CPU use > 80% AND user zone = "near covered zone" => ComponentDestination = nearest node to the user
- disconnectedMode = "true" AND user terminal CPU use < 80% => ComponentDestination = UserTerminal
The constraint1 and the constraint2 can be satisfied in the same time since they are related to different types of context. When such rules are met, their context constraints must be completed by the negation of all the other constraints associated with the other rules able to occur at the same time. Thus, the previous example must be described as in Figure 8.

```
constraint1: user zone="near non covered zone" ∧ user terminal
CPU use<80% ∧ loggingMode="false" =⇒ user terminal
constraint2: loggingMode="true" ∧ user terminal CPU use>80% ∧
user zone="near covered zone" =⇒ the nearest node to the user
constraint3: loggingMode="true" ∧ user terminal CPU
use<80% =⇒ user terminal
```

Fig. 8. Placement of the Logging Component

Sometimes none of the rules associated with a deployment parameter is able to be applied because none of their context constraints is satisfied, this case does not present an inconsistency, for the following reasons:

- If the parameter presents the existence of a component instance or a connection, it means that the context state does not require their deployment.
- If it presents a component placement or an implementation version, it means that the component will be placed automatically according to the context and the resources availability (cf. Section VII).
- If the parameter presents a value of a configuration property, the property will have its default value.

B. Consistency of the Application’s Architecture

There are two types of inconsistencies in the architecture rules at the application level: the detachments of obligatory components and the detachments of optional components.

To be sure that the application is functional when when the optional components are not deployed, we have to check that there is no detachments between the obligatory components of the application.

The set of obligatory components and the connections between them form an oriented graph where the nodes of the graph represent the components and the arcs represent connections between them. As a connection is a link between a component which provides an interface and a component which uses this interface, an arc is always oriented towards the component which uses the interface.

To check that there are no detached components, we check that this graph is related. For that, we apply an exploration algorithm of oriented graph in-depth first [7]. If the algorithm is able to explore all the graph nodes, there is no detachment between the obligatory components of the application. In the contrary case, the components corresponding to the not explored nodes are indicated to the deployment planner as detached components.

After checking the consistency of the connections between the obligatory components, the checking tool checks the deployment consistency of the optional components and the connections between them. To do that it checks that all the context constraints which condition the ways of each optional components towards obligatory components also condition the existence of this component (a way is a set of connections).

There are other types of inconsistencies, which are related to the description of the deployment parameters such as specifying a connection between two components, which have incompatible ports but in this paper we focus on the inconsistencies related to the adaptation rules.

VII. DEPLOYMENT ADAPTATION ALGORITHMS

To make the deployment of a component-based application suited to the context, the deployment adaptation algorithms have to carry out the following stages:

- Determination of the application architecture
- Determination of the placements and the implementation versions of the component instances
- Determination of the values of the configuration properties of the component instances.

These stages will be detailed in the following.

A. Application Architecture

Let inst a component instance and conn a connection specified by the application assembler. The algorithm which determines the architecture of the application is the following:

```
For each (inst) do
  If (Obl(inst)=true) Then put inst in the FDP
  Else If (Exist(inst)=true)
    Then put inst in the FDP
For each (conn) do
  If (Obl(conn)=true) Then put conn in the FDP
  Else If (Exist(conn)=true)
    Then put conn in the FDP
```

Obl(inst) checks if a component instance is obligatory or not and Exist(inst) checks if the existence condition of an optional inst is satisfied by the context or not. FDP is the Final Deployment Plan

B. Component Placement and Implementation

The placement algorithm processes the placement rules at the application level then it determines the placement site and the implementation version of all the component instances for which a site has not been specified at the level of the application rules. To perform the second stage, the placement algorithm starts by finding the valid implementations/nodes assignments for each component instance. As we can have several valid assignments for the same component, the algorithm selects the best one. The best assignment can be chosen according to several different criteria such as maximizing the number of satisfied weak constraints or optimizing the resources consumption. We use the A* algorithm [5] in order to explore all the possible assignments and approach the best one.

Let consider the following functions:
• ApplicationLevelRules(deploymentParameter) checks if there are application level rules which are associated to a given deployment parameter.
• ApplicableApplicationLevelRule(deploymentParameter) checks if one of the application level rules constraints associated with a given deployment parameter is satisfied by the context.
• ApplicableComponentLevelRule(deploymentParameter) checks if one of the component level rules constraints associated with a given deployment parameter is satisfied by the context.
• FindPossibleImplementationAndNodes(inst) finds the valid implementations/nodes assignments of a component instance. It extracts the component level rules of the implementations versions choice of the component. Then, it finds a placement node for each of its implementations i.e. a node which satisfies its strong constraints.
• DetermineParallel() determines the set of components which can be executed in parallel and whose placement was not assigned.

The algorithm which determines the placements and the implementation versions of the component instances is the following:

\[
\text{For each (inst) in the FDP do}
\]
\[
\text{If (ApplicationLevelRules(placement) = true)}
\]
\[
\text{Then If (ApplicableApplicationLevelRule(placement) = true)}
\]
\[
\text{Then AssignPlacement(instance)}
\]
\[
\text{Else FindPossibleImplementationAndNodes(inst) Parallel=\text{DetermineParallel}()}
\]
\[
\text{For each set of components } \subseteq \text{Parallel, apply the A* algorithm:}
\]

While at least one component is not placed, do:

1) Choose one of the components which are not placed.
2) For each node where the component can be placed, do the following:
   a) Estimate the additional nodes resources by supposing that the components which are not placed yet, are placed on the nodes which maximize the additional nodes resources.
   b) Estimate the number of satisfied weak constraints by supposing that the components which are not placed yet, are placed on the nodes which maximize the number of satisfied weak constraints.
3) Select the assignment which maximizes the additional nodes resources and the number of satisfied weak constraints.

More details about the components placement algorithm, the optimization of the resources consumption, and the calculation of the number of satisfied weak constraints are given in [6].

C. Component Properties Values

The algorithm which determines the values of the configuration properties of the component instances is the following:

\[
\text{For each (inst) in FDP do}
\]
\[
\text{For each property of inst}
\]
\[
\text{If (ApplicationLevelRules (property) = true) Then}
\]
\[
\text{If (ApplicableApplicationLevelRule(property)=true)}
\]
\[
\text{Then assign the component property value}
\]
\[
\text{Else}
\]
\[
\text{If (ApplicableComponentLevelRule(property)=true)}
\]
\[
\text{Then assign the component property value}
\]
\[
\text{Else propertyvalue=defaultValue}
\]

After these stages are carried out, a deployment plan can be generated. This plan contains the decisions made by the deployment adaptation algorithms about the different deployment parameters of the application to be deployed.

VIII. IMPLEMENTATION AND EVALUATION

The deployment adaptation rules presented in this paper are processed by an entity called a DeploymentAdapter that we developed on top of a CCM deployment tool. The platform we used was OpenCCM [7]. A context infrastructure carries out the context information collection from the different context sources and filters the relevant contexts for the deployment of the different applications. For more details about the context infrastructure refer to [8].

A. Usability and Applicability of CADeComp

Just-in-time deployment assumes the existence of a networking infrastructure from where the software should be installed. This infrastructure is generally offered by a deployment provider. This assumes that the user must be able to connect to this infrastructure at the deployment time. In case that all the components are installed on the user terminal, this connection does not need to be maintained during the application use. The networking infrastructure is dynamic, the context state of its nodes is collected and updated at each deployment.

Context-aware Just-in-time deployment requires that the application is deployed for a given use and a given context. This supposes that the context state collected at the deployment time remains acceptable during the execution of the application.

B. Validation Tests

We made a validation test of the deployment of the online shopping application with CADeComp.

During the deployment tests, the DeploymentAdapter was run on Red Hat Linux 9.0 powered by a 1.8GHz Pentium Intel with 256MB of RAM. The PDAs used are iPAQ with Microsoft Pocket PC 4.20.0 powered by a TI OMAP1510 processor and 56MB of Memory. Besides the PDAs, we have four nodes which can contain the components when the users terminals do not have enough resources. These nodes play the role of the component servers offered by the deployment providers. On each of these nodes we instantiated a DB component which contains different goods. Each of these servers is associated with a different zone (zone1, zone2, zone3...).
and zone 4). Zone 4 is a non-covered zone thus zone 3 is considered as near non-covered zone.

To perform the validation tests we considered several different values that can be taken by the contexts that impact the deployment of the online shopping application. These contexts are the following:

- The user preferences.
- The network connection.
- The user location.
- The resources provided by the user terminal and the servers offered by the deployment provider.

The user has a deployment client which allows him to initiate the deployment and to enter his preferences: if he wants to work in a disconnected mode or not, the language he wants to use and the displaying colors he prefers. The location of the users was simulated by an application which gives randomly the zone where the user is located.

We tested the deployment of the application with different context values. All the expected results matched with the obtained deployments. For example, when the location of the user changes, the GUI and the Cart components are connected to the DB component which corresponds to the location zone of the user. When the user is in zone 3, the LV component and its connections to the DB component and the GUI component are deployed.

These validation tests show that apart of their consistency, the adaptation rules are able to provide an application which is in accordance with what the user would expect in terms of functionality.

C. Performance Measurements

We have performed experiments in order to study the additional delays caused by the adaptation rules processing. Figure 9 shows the average deployment time with and without adaptation according to the number of deployed components. Each test was executed 20 times in order to obtain meaningful averages. The average variance of these measurements is equal to 0.76.

Fig. 9. Average Deployment time according to the Number of Deployed Components

The results show that the adaptation time increases linearly with the application size at an average of 0.21%. The processing of the adaptation to the context represents an average of 5.48% of the total average deployment time.

Since we have a number of adaptation rules associated with each component to deploy at the application and the component level, the number of rules increases with the number of components involved in the application. Which explains the delay increase corresponding to the increasing number of components.

IX. RELATED WORK

A wide variety of technologies already exist to support different deployment activities such as installation, configuration, packaging and un-installation of applications [9] [10]. These technologies are well described in [11] but the majority of them are limited to the deployment of monolithic applications which are deployed on a single site.

The software Dock [12] is an agent-based framework which supports the entire deployment activities of distributed applications. The framework is enabled by the use of a standardized deployment schema for describing both software systems and sites. The software Dock does not support the deployment of component-based applications that are described as an assembly.

Several component-based deployment solutions have been specified, such as the CCM (CORBA Component Model) component packaging and deployment model [1], the EJB deployment solution [13], the .Net deployment solution [14] and the J2EE deployment API [15]. The deployment process of these solutions is not context-aware and is not completely automated. It requires the user’s intervention in the placement of the components.

OSGi [16] is an open specification for a component model that allows services to be deployed and managed. OSGi is a good starting point for developing flexible applications that are easily manageable but the specification does not describe anything about the adaptation of the deployment process to the context.

[17] addresses the deployment of component-based personalized services. These services are user-centered applications whose behavior must be tailored according to the user’s preferences. This paper has introduced the personalized deployment that is an extension of the CCM deployment solution for taking into account user preferences at deployment time. It does not address context-awareness but rather limits its study to the user’s preferences. Moreover, it is specific to the CCM deployment solution.

[18] describes a deployment approach which supports software deployment into networks of distributed, mobile, highly resource constrained devices. This approach is based on the principles of software architecture, which allow the description of a software assembly. However, the application architecture is not able to vary according to the context.

Several component-based deployment solutions support only the context, which is the deployment target environment,
and do not consider the user context, which consists of the user’s preferences, activities and physical environment. As examples of these deployment solutions, we can cite:

- The OMG Specification of the Deployment and Configuration [3], which is a data model for the description of the artifacts that are part of the deployment, of the creation of component instances from artifacts, of the placement of these instances, and of information about connections between them. This specification also represents a data model for the description of the domain into which applications can be deployed as a set of interconnected nodes.

- The COACH deployment specification [19], which involves a set of descriptors for describing the topology and properties of the target environment and an assignment tool for assigning the proper implementation code of an assembly to the right target nodes meeting the component requirements.

- The partitionable services framework [20], which enables services to be flexibly assembled from multiple components. The central feature of this framework is its ability to adapt to different network conditions by providing a planning module which determines the appropriate locations in the network to instantiate the services.

The solution that we propose in this paper is component-based because component-based applications can be adapted more easily than traditional applications by simply adding and deleting components. Our solution is also able to consider different types of context and is not limited either to the target environment or to the user preferences.

In order to deploy the assembly of the components making up the application, the solution that we propose extends the distributed component-based deployment tools to support the context-awareness. The originality of our solution consists in the use of a methodology based on the variation of the deployment parameters and which determines the assembly of the applications according to the context.

X. CONCLUSION

Adaptive deployment is a key factor for the deployment in mobile environments. This paper defines a methodology which allows the adaptation of the deployment of component-based applications according to the context. This methodology is based on a set of adaptation rules which are processed by a set of algorithms. These adaptation algorithms are able to vary the different deployment parameters of component-based applications according to the context. The adaptation rules are divided into two levels: The rules which are specific to the component and those which are specific to the application. This division allows the consideration not only of the component semantics but also the semantics of the application into which the component is integrated.

Our methodology makes classical deployment tools able to satisfy the deployment requirements of mobile users given in Section 2. The adaptation rules are processed just-in-time and enable the deployment tool to be context-aware, to be adaptive, and to save the hardware resources. The rules consistency is checked by a checking tool.

The deployment adaptation and automation does not add important delays compared to classical deployment times without adaptations.

We mapped our deployment model on CCM because it provides the most rich deployment tool. In further phases we will map our model on other platforms such as .Net or EJB/J2EE.

The methodology specified in this paper adapts the initial deployment of new applications to the context. In further phases, we intend to extend this methodology to support the dynamic reconfiguration of previously deployed applications. Dynamic context-aware reconfiguration is more difficult than the context-aware initial deployment because of the intrusive characteristic of a reconfiguration process that could threaten the application’s integrity.

Our solution is based on just-in-time deployment, which leads to the un-installation of the application just after its deactivation in order to save the node’s scarce resources. This solution is based on the hypothesis that mobile users do not always use the same applications because their context, and consequently, their requirements are in constant change. But if the users frequently need to use the same application, another solution can consist in using different policies to un-install the deployed applications only when we need resources, such as the LRU (Least Recently Used) or the FIFO (First In First Out) policies. These policies add adaptation delays but it is worth studying different policies in the search for a policy that adds acceptable delays.

REFERENCES


