Available online at www.sciencedirect.com
ScienceDirect
Journal of Network and Computer Applications 31 (2008) 224–257
Journal of NETWORK and COMPUTER APPLICATIONS
www.elsevier.com/locate/jnca

CADeComp: Context-aware deployment of component-based applications

Dhouha Ayed[a,*], Chantal Taconet[b], Guy Bernard[b], Yolande Berbers[a]

[a]*Department of Computer Science, K.U. Leuven, B-3001 Leuven, Belgium*
[b]*GET/INT, CNRS Samovar 5157, 9 rue Charles Fourier, 91011 Évry, France*

Received 24 July 2006; accepted 4 December 2006

Abstract

The expansion of wireless communication and mobile hand-held devices makes it possible to deploy a broad range of applications on mobile terminals such as PDAs and mobile phones. The constant context changes of mobile users oblige them to carry out many deployment tasks of the same application in order to obtain an application whose configuration satisfies the context requirements. The difficulty and the frequency of these deployment tasks led us to study the deployment in a mobile environment and to look for a solution for the automation of the deployment adaptation to the context. This paper studies the deployment sensitivity to the context in order to identify the variable deployment parameters and to analyze the impact of the deployment adaptation on the production life cycle of applications. The contribution made by this paper consists in an innovative middleware entity called Context-Aware Deployment of COMPonents (CADeComp), which can be plugged into existing middleware deployment services. CADeComp defines a flexible data model that facilitates the tasks of component producers and application assemblers by allowing them to specify the meta-information required to adapt the deployment to the context. The advantage of CADeComp is that it is based on reliable adaptive mechanisms that are defined by a platform-independent model according to the MDA approach. We propose a mapping of the CADeComp model to CCM. CADeComp was implemented and evaluated on this platform.

Keywords: Deployment; Components; Context-awareness; Distributed applications; CCM

*Corresponding author.
E-mail addresses: Dhouha.Ayed@cs.kuleuven.be (D. Ayed), Chantal.Taconet@int-evry.fr (C. Taconet), Guy.Bernard@int-evry.fr (G. Bernard), Yolande.Berbers@cs.kuleuven.be (Y. Berbers).

1084-8045/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jnca.2006.12.002

1. Introduction

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. Banking applications, games and multimedia services are examples of applications that are used by mobile users.

Thanks to this progress, it is becoming possible to include the mobile devices in complex distributed systems. There are two main reasons why these distributed systems cannot be assimilated to the fixed distributed systems. The first one is that the resources of mobile devices remain scarce compared to those of traditional terminals: mobile devices are equipped with a slower CPU and a limited memory size, they seldom have a secondary memory in their initial configuration and they can have low battery power. The second reason is that mobile distributed systems are dynamic: users undergo constant changes in their context, such as a high variability of network bandwidth, location and physical environment, while fixed distributed systems are characterized by a stable environment (large bandwidth, fixed locations of users and hosts).

The dynamism of mobile systems has created new challenges in various fields of computer science. These challenges mainly consist in making the services context-aware, which means making them able to detect context information, such as the user location, his preferences and his hardware environment and to adapt according to the changes in this context. These challenges require the review and extension of several solutions used in fixed distributed systems. Software deployment, which consists essentially in installing software and making it available to users, is one of the services that must be made context-aware. The idea of extending the existing deployment solutions to be adaptive is motivated by the diversity of the applications which are deployed by mobile users and the frequency of these deployments.

A mobile user often needs to be able to deploy the same application in 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 and on a friend's laptop when he is not at home. These various devices have different hardware and software environments. Hence, the deployment needs to be processed differently according to the environment and the user's condition.

The resource limitation of hand-held devices and the high variability of contexts lead mobile users to perform several repetitive deployment activities: (1) The user has to install his applications on all the different types of devices. (2) If the user's device does not have a secondary memory, then the applications have to be redeployed each time they are required. (3) If the user's device has a secondary memory, this memory generally has a limited size that is not sufficient to install all the applications needed by the user at the same time. 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.

These repetitive tasks of installations, configurations and uninstallations, which can be difficult, complex and time consuming for the users, can limit the use of mobile applications. The difficulty and the frequency of these deployment tasks led us to study

software deployment within a mobile environment and consequently to identify the following requirements for a deployment tool:

- *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. In order to adequately deploy the required applications, it must take into account the current resources of the user's device, as well as the current user's activity, his preferences and his physical environment.
- *Just-in-time deployment*: The applications must be deployed only when accessed by the user and undeployed automatically just after their deactivation. Just-in-time deployment is necessary to make deployment context-aware. Furthermore, the deployment of several applications on the user's device reduces the available memory on the device and slows down the execution of the newly deployed applications. Thus, just-in-time deployment contributes to saving device resources.
- *Adaptive just-in-time 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.

In this paper, we study the deployment sensitivity to the context and we propose a solution to adapt the deployment to the context. This solution consists in an innovative middleware entity called Context-Aware Deployment of COMPonents (CADeComp), which satisfies the requirements mentioned above. CADeComp extends existing deployment services by giving them adaptation capabilities. It allows component producers and application assemblers to specify the meta-information that is necessary to adapt the deployment to the context. This meta-information is defined by a flexible data model and processed by a reliable execution model that specifies generic adaptive entities. One important feature of CADeComp lies in the fact that it is platform independent. We focus on component-based applications since they are flexible, a fact which makes them more suitable in variable mobile environments.

The paper is organized as follows. Section 2 presents a background and discusses related work. Section 3 studies the sensitivity of the deployment to the context. Section 4 introduces the methodology we used to adapt the deployment to the context. The CADeComp data model and execution model are specified in Sections 5 and 6, respectively. Section 7 discusses the implementation and the experimental results before the paper concludes in Section 8.

2. Background and related work

The idea of proposing an approach to adapting the deployment to the context was justified by the total lack of any deployment service intended to be used in a mobile environment, while many context-aware systems and mechanisms exist in other fields. In this section, we present existing context-aware infrastructures and mechanisms. Then, we

introduce some principles of component-based applications and existing deployment services.

2.1. Context-aware middleware

Context-aware applications are able to collect context information and quickly adapt their behavior to context changes. To achieve such a level of awareness, applications need to periodically query heterogeneous physical sensors. Building applications by directly accessing context information is expensive and error-prone, and it leads to non-portable applications. For this purpose, a middleware is required to provide abstractions for the fusion of sensor information and to implement the adaptation mechanisms.

Context information, which can be acquired from heterogeneous and distributed sources (sensors, files, applications), may be dynamic and may require an additional interpretation in order to be meaningful for an application. A set of architectures for acquiring context information can be found in the literature. For instance, SOCAM (Gu et al., 2004) uses a central context interpreter, which acquires context data through distributed context providers and offers them in a processed form to the clients. CoBrA (Chen, 2004) is an agent-based architecture for supporting context-aware computing in so-called intelligent spaces. Intelligent spaces are physical spaces (e.g. living rooms, vehicles and meeting rooms) that are populated with intelligent systems that provide pervasive computing services for users. CASS (Fahy and Clarke, 2004) listens for updates from distributed sensors, stores the gathered information in a database and uses an interpreter and a rule engine to interpret context.

The intelligence of a context-aware system can be increased by using an inference engine, which reasons on context information. An inference engine is implemented by a set of facts and rules applied to context information collected from several sources. Among the existing implementations of inference engines, we can mention Flora 2 (Chen et al., 2003) and Jena 2 (Jena2). CORTEX (Sorensen et al., 2004) uses an inference engine to reason on context.

The most consistent attempt at developing a reusable solution for context acquisition is context toolkit (Dey et al., 2001; Dey, 2001). This is a generic framework that enables rapid prototyping of context-aware applications by using a set of abstract components (widgets, interpreters and aggregators). These components support the acquisition of context data from sensors and their processing into high-level context information.

Context-aware middleware are not limited to context acquisition, aggregation and interpretation. A lot of them allow the developers to implement different adaptation mechanisms in order to adapt applications to the context. Among the existing adaptation mechanisms we can mention reflection, aspect-oriented programming and contracts:

- Reflection is the capacity of a system to observe and to modify itself (Bobrow et al., 1993). The observation is the mechanism of introspection and the modification is the mechanism of intercession. CARISMA (Capra et al., 2003), K-components (Dowling and Cahill, 2001), ReMMoc (Grace et al., 2003), OpenORB (Blair et al., 1998) and CORTEX (Sorensen et al., 2004) are examples of reflective context-aware middleware.
- Aspect-oriented programming enables developers to build applications by separating their functional from their non-functional codes. These aspects are combined thanks to the concept of pointcuts and weaving (Kiczales). To develop adaptive applications using

this approach, the adaptation process is implemented as a non-functional code. RAM (David and Ledoux, 2002) is an example of adaptive middleware based on AOP.

- Contracts enable developers to specify constraints to be satisfied by context so that the application behaves as expected (Meyer, 1992). They also allow developers to describe adaptation policies that are not hard-coded into the adaptation process, but are rather described into a file named "adaptation contract". This file is interpreted by an adaptation process and can be loaded and unloaded at runtime.

Context-aware middleware present a set of reusable mechanisms for the collection, processing, interpretation and presentation of context information. They can also provide reusable adaptation methods based on different mechanisms. Context-aware middleware separate context processing from the system to be adapted in order to dissimulate the complexity of this processing. They shorten and facilitate the various phases of applications development, since applications directly reuse the mechanisms offered by the middleware without redeveloping them.

2.2. Component-based applications and existing deployment services

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties" (Szyperski, 2002). A component represents a black box concerning which no detail about its implementation is known from outside; only the component interfaces are public. A component-based application is a collection of independent software components connected to each other through their ports.

We distinguish between a component instance and a component type. The latter is a reusable static entity that only exists at design time and contains a complete description of the type of a component in terms of its interfaces, its interaction modes and its attributes. Component instances are runtime constructs containing a certain state. For the remainder of the paper we reserve the term component for a component instance. The implementation of a component type represents its functional code. Its non-functional code is managed by the component-based middleware.

A component may have two types of properties: non-functional properties which are properties concerning quality of service such as the persistence of a component, or its security policies and functional properties which are the configuration attributes of a component.

Components need a component middleware which manages in a transparent way their life cycle (creation, deletion, discovery), their distribution, the non-functional services they need and their deployment. For example, CCM (OMG, 2002) defines a model of a component middleware for CORBA components, and J2EE (Sun Microsystems, 2003) is a component middleware for EJB components (Sun Microsystems, 2002). The .NET framework (.Net framework) manages the life cycle and the non-functional services of .NET components.

The deployment services provided by CCM, J2EE and .NET as well as other existing deployment models such as the OSGi (Open Services Gateway Initiative, 2003) deployment service are based on a static approach. In this approach, whatever the context, the architecture of the application is fixed 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.

More sophisticated deployment solutions have been proposed. We can mention (Sabri, 2003) which 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. The 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.

Several component-based deployment solutions support context in the sense of 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 mention the specification of deployment and configuration of component-based applications called D&C (OMG, 2006), the COACH deployment specification (IST, 2003) and the partitionable services framework (Ivan et al., 2002).

The goal of our work is to introduce existing context-aware mechanisms in the deployment process and to extend this process in order to support adaptability to different types of contexts.

3. Ability of the deployment to be context-aware

In this section, we study the characteristics, the steps, the activities and the meta-information of component-based application deployment in order to identify the deployment elements that are able to vary according to the context. The deployment activities and meta-information presented in this section have been defined after studying and comparing all the models of the deployment services we mentioned in Section 2.2 (Ayed, 2005).

3.1. Deployment life cycle of component-based applications

Deployment refers to all the activities performed after the development of the software which make it available to its users. The deployment activities consist essentially in installing and activating the software but can also include release, deactivation and uninstallation of the software. Since only the deployment life cycle of monolithic (non-component-based) applications has been defined in previous works (Hall et al., 1999), we try in the following to provide a definition for the deployment life cycle of component-based applications by showing the characteristics of their deployment activities.

The release activity is the bridge between the development process and the deployment process. It encompasses the activities needed to package and advertise the software for deployment to user sites. The release of a component-based application consists in releasing its components than releasing the application itself. The release of a component is carried out by a *component producer* who produces a component package. This package contains a description of the component type, its implementation and its dependencies. The *release* of a component-based application is carried out by an application *assembler* who analyzes the compatibility of the components produced by different component producers and assembles an application. The application *assembler* builds a component

assembly package that contains several component packages as well as meta-information describing this assembly (see Section 3.2).

The *installation* activity must configure all the resources necessary to use a given software. It includes the unpacking of the component assembly package of the application and the unpacking of the component packages.The installation activity is carried out by a *deployer*, who follows the instructions specified by the application assembler and the component producer of each component.

The *activation* starts the execution and thus enables the use of the deployed software. For a simple tool, activation involves establishing a command for executing the software binary. The activation of a component-based application involves the activation of its overall components by *instantiating* them and *connecting* them. The *deactivation* activity shuts down the software execution by shutting down its overall components.

The *uninstallation* activity undoes all the changes to the site that were caused by previous installation activities for a given software. This activity must examine the current state of the site, its dependencies and constraints, and then remove the overall components of the software system in such a way that it does not violate these dependencies and constraints.

The definition of the deployment activities is very important because it enables us to identify where the context-awareness process will be integrated into the deployment life cycle and who the deployment actors are that play a role in the deployment adaptation.

3.2. Deployment meta-information

In addition to the component-based deployment activities, the study we carried out in (Ayed, 2005) helped us to identify the meta-information generally required by the deployment services to achieve the deployment activities. We distinguish meta-information related to each component of the application (placed in the component package) from meta-information which describes and characterizes the general component assembly of the application (placed in the component assembly package).

The meta-information related to the *deployment of a component* is the following:

- *General meta-information*: Represents general descriptions about the component package, such as its name, its version and its license.
- *Structural meta-information*: Defines the type of the component, i.e. its interfaces and its attributes.
- *Dependencies meta-information*: Specifies the dependency of the component type on other packages and resources such as a database or a file.
- *Implementation meta-information*: Describes an implementation of the component. If the component has several implementations, then specific information about each one must be provided, such as the location of the implementation (for example, a URL from which someone can download it), the programming language used and the compiler used.
- *Non-functional properties meta-information*: Assigns values to quality of service properties of the component such as its persistence or security properties.
- *Functional properties meta-information*: Specifies the values that will be taken by the configuration attributes of the component.

The meta-information related to the *deployment of an application* consists in an *application deployment plan*. This meta-information describes the *architecture* of the application, i.e. the component instances that make up the application and connections between them. It also describes all the information necessary for the deployment of each one of these instances: their *placement* and configuration. The application deployment plan is used as a guide for the deployment of the application.

In current component-based middleware, this deployment meta-information is static and does not take into account context information.

3.3. Illustration with a disaster scenario

Let us consider a scenario that we have proposed in the RNTL AMPROS[1] project. This scenario involves a large-scale disaster, such as an aircraft or train crash, and includes a large number of victims and several rescue teams.

In this scenario, the rescuers use PDAs and a mobile network infrastructure to share information about victims, communicate with other team members, send medical diagnoses, determine hospital availability, etc. All the needed equipment, such as medical equipment, computer servers and network infrastructure, is placed in the ambulances that are deployed in the disaster area. The rescuers' devices are unable to contain all the applications that perform the needed tasks at the same time. They consequently require a just-in-time deployment, which deploys the applications when they are needed and takes into consideration the rescuers' requirements, as well as the situation both of rescuers and victims.

In this paper, we illustrate the context-aware deployment with the deployment of an application called "disaster management". This application allows rescuers to communicate the victims' conditions to other rescuers belonging to different teams. Fig. 1 shows some possible architectures and placements of the disaster management application that vary according to the context. This application contains the following components:

- A graphical user interface (GUI) component, through which the victims' conditions or diagnoses can be entered. This component must be instantiated on the rescuer's device.
- A victim manager component, which receives information about all the victims' conditions, processes it and sends it to the concerned partners. These partners represent the group of rescuers working with the rescuer who entered the victims' conditions. The placement of this component depends on the availability of the resources; it can be placed on the rescuer's device or on the least loaded server of the disaster area in order to reduce the device load.
- A logging component, which logs the actions of a component to which it is connected. A logging component is deployed in only two cases: (1) when the rescuer is close to a non-covered network zone (the logging component will log the victims' conditions and will send them to the victim manager when the rescuer device will be reconnected). (2) When the rescuer prefers to log the victims' conditions. When this component is deployed, it is connected to the GUI component and the victim manager component.
- One or several view components, which enable the partners to receive and display information about the conditions of the victims. The number of deployed components

[1] http://www-inf.int-evry.fr/AMPROS.

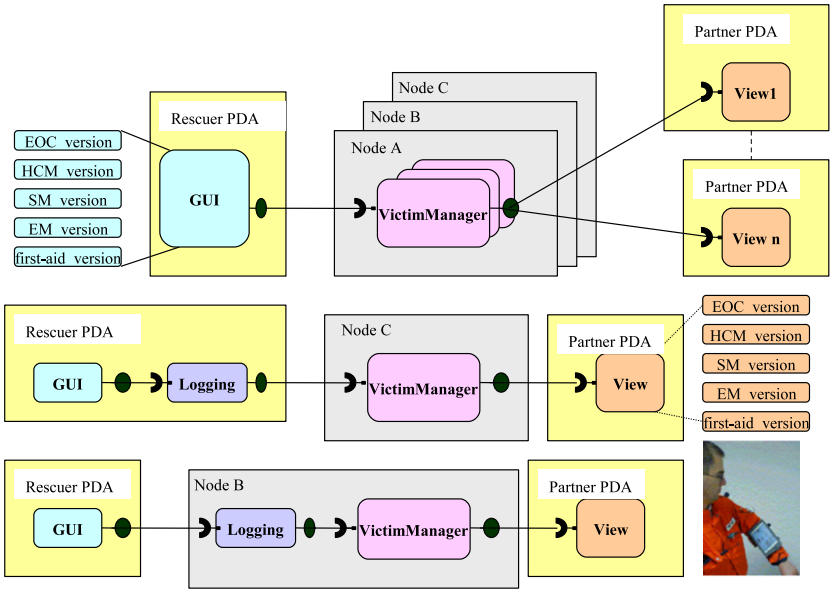

Fig. 1. Possible architectures and placements of the partners' application.

of this type depends on how many partners the rescuer who enters the conditions of the victims wants to collaborate with. This component must be placed on the device of each of the concerned partners.

The GUI and the view components have several implementation versions, which do not offer exactly the same functionalities. Each implementation version is dedicated to a different rescuer profile (the choice of a version at the deployment time depends on these profiles). The profiles of the different rescuers are as follows: (1) The emergency operation commander (EOC) coordinates all rescue operations. (2) The healthcare manager (HCM) organizes emergency operations. (3) The sorting manager (SM) sorts the victims according to the seriousness of their injuries. (4) The evacuation manager (EM) evacuates victims to hospitals according to the diagnoses. (5) The first-aid workers provide first aid for victims.

We will use this application scenario to illustrate the deployment context-awareness and the solution that we propose.

3.4. Context sensitivity of deployment

We argue that the adaptation of the deployment according to the context does not require the adaptation of the deployment activity mechanisms themselves such as the installation, instantiation or connection mechanisms. Context changes rather impact the deployment meta-information. For example, if two people want to deploy the same type of component with different languages, the deployment tool will instantiate the component for each of them with a different value of the functional property representing the language used. In this case the deployment tool uses the same instantiation mechanism for both components, but with different values of meta-information.

Among the meta-information we mentioned in Section 3.2 we identify six meta-information categories that can vary according to the context and that we call *deployment parameters*:

- *The architecture of the application*: The number of the component instances of an application and the number of connections between them can vary according to the context. The component assembly package of the application may contain a description of several component instances and connections, but only the component instances and the connections required by context must be instantiated. For example, the architecture of the disaster management application depends on the network connection, the number of partners and the rescuer preferences. The network connection and the rescuer preferences determine whether the logging component will be deployed or not, and the number of partners determines the number of view components to be deployed.
- *The component instances placement*: If the resources of the user device are insufficient to instantiate all the components of an application, only one part of these components will be instantiated on the user device and the remainders will be distributed on distant nodes. These nodes can be selected according to several contexts, such as the availability of their resources and the localization of the user. For example, as illustrated in Fig. 1, the placement of the victim manager component depends on the availability of resources.
- *The component implementations*: Since each type of component can have several implementations, one of these implementations has to be chosen during the instantiation of the component. This choice depends primarily on the software and hardware platform where the components will be instantiated, but can also depend on other types of contexts. For example, the choice of the implementation version of the GUI component depends on the rescuer profile.
- *The component functional properties*: Each functional property of a component can depend on context. For example, the property which represents the language of the user in the GUI component depends on the language the rescuer prefers to use.
- *The component non-functional properties*: Each non-functional property of a component can depend on context. For instance, the security properties of the victim manager can depend on the rescuer requirements.
- *The component dependencies*: When the context of use of the component changes, its dependencies can change. For example the victim manager component has a dependency on a database which contains the records of the victims. This dependency is required only if the state of a victim necessitates to access his records.

The general meta-information and the structural meta-information of a component do not vary according to the context. The structural meta-information is only used to check the compatibility between the components' ports.

The deployment parameters we have identified in this section can vary according to different types of context. This variation depends on the semantics of the application to be deployed.

4. An approach for adapting the deployment to the context

According to Section 2.1, we can choose from three adaptation mechanisms to adapt the deployment to the context: reflection, aspect-oriented programming and contracts. As the adaptation of the deployment to the context does not require the modification of the deployment mechanism (see Section 3.4), we do not need to use reflection because it would allow the deployment service to adapt itself. We also avoided the aspect-oriented programming because it would change the deployment mechanisms and would be complicated to implement and to debug. We consequently choose to use contracts. A deployment contract between an application and the context describes the possible variations of the application deployment behavior and the context constraints necessary to carry out these variations. The variations of the deployment behavior are expressed through the variations of deployment meta-information.

The deployment contracts depend on the semantics of the applications. Since the different producers of the applications are able to determine the requirements of the application deployment in terms of context information, the contracts must be written during the production process of the applications.

As the release of a component-based application consists in releasing its components and then assembling them, we propose to specify these contracts during the release of the various components of the application (by the component producer, see Section 3.1) and during their assembly (by the application assembler, see Section 3.1). For example, the component producer can specify the platform constraints necessary for the component placement. The application assembler can define the variability of its architecture according to the user preferences.

The goal of the CADeComp middleware entity we propose in this paper is to ease the role of component producers and the application assemblers by providing simple means to write deployment contracts and define adaptation mechanisms that will process these contracts.

The deployment contracts play a significant role in the automation and the adaptation of the deployment process to the context. The CADeComp adaptation mechanisms will use these contracts to determine the values of the deployment parameters just-in-time, after taking into account the context state. Consequently, three of the requirements that we mentioned in the Introduction will be satisfied: the deployment automation, the adaptive deployment and the just-in-time deployment.

Since we want to deploy component-based applications, CADeComp is consequently integrated into a component middleware which manages the life cycle, persistence, security policy and deployment of the components. Our objective is not to invent a new deployment mechanism, but to extend existing deployment mechanisms in order to support context-awareness. This is why we aim to design CADeComp so that it can be placed above existing deployment models, such as those provided by J2EE, CCM and .NET.

We showed in Section 2.1 that a context-aware middleware separates the context processing from the system to be adapted, it dissimulates the complexity of the context processing and it shortens the various phases of software development. For these reasons, we place CADeComp above a context-aware middleware (see Fig. 2).

Although deployment parameters may be closely related to the deployment tool used, and context information format depends on the context-aware middleware used, we want the description and the processing models of the contracts to be independent of the

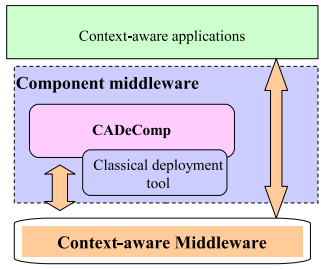

Fig. 2. CADeComp: architectural overview.

component middleware and the context-aware middleware. For this purpose, we define a platform-independent deployment adaptation model for CADeComp. This model follows the model driven architecture (MDA) approach of OMG (OMG, 2003) in order to be generic and to be able to extend several deployment tools. The definition of a platform-independent model (PIM) allows us to reuse this model for different platforms by automatically generating several platform-specific deployment adaptation models.

The CADeComp deployment adaptation model consists of a data model which allows component producers and application assemblers to specify deployment adaptation meta-information and an execution model which defines the generic entities that process the deployment adaptation. The data model and the execution model are detailed in the following sections.

5. CADeComp data model

In this section, we propose a platform-independent data model for the specification of meta-information that is necessary for the adaptation of the deployment to the context. This meta-information represents the deployment contracts. One of the possible solutions for writing these contracts is to associate to each context state all the possible variations of the deployment behavior together, as we proposed in (Ayed et al., 2003). This description is made for all the components of the application at the same time without any consideration of the types of the deployment parameters. This solution can be very complicated for the actor who will specify these contracts, and it can result in him forgetting the specification of some possible variations. This is because it is difficult to imagine the variations for different types of parameters at the same time. The processing of these contracts would also be difficult. For this reason, we propose to divide this meta-information between a component level and an application level. We also propose that the deployment contracts at the component and application levels be directed by the types of the deployment parameters. This structure provides more flexibility and independency between component producers and application assemblers. It also helps these actors to write more complete and efficient contracts. Fig. 3 shows an overview of deployment adaptation meta-information that we model in this section.

Both levels of meta-information (component and application levels) use context information in order to specify the variation of the deployment parameters. In order to follow this two-level structure, we start this section by presenting the *context model* common to both levels, and then we detail the *component package model* which defines the

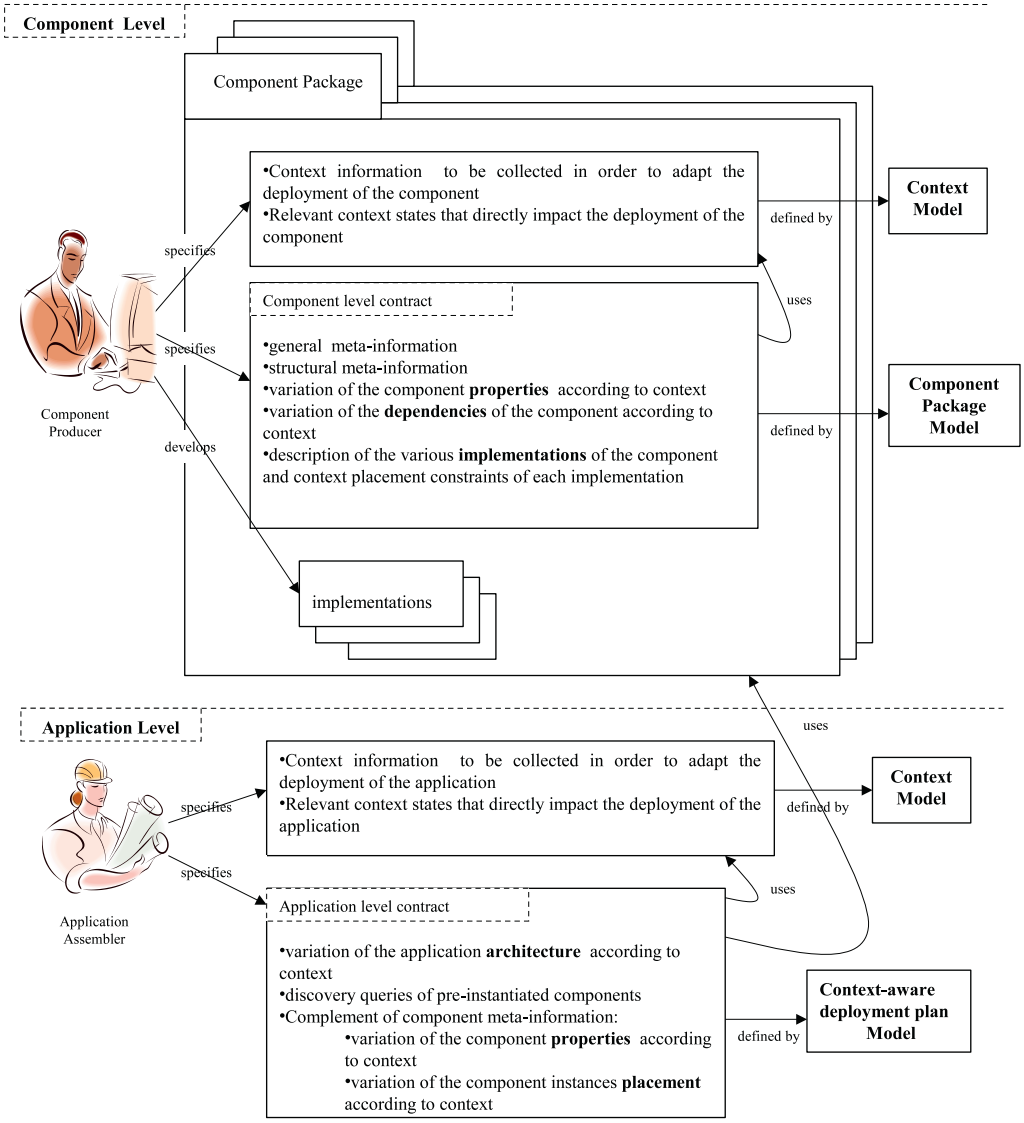

Fig. 3. Overview of deployment adaptation meta-information.

meta-information at the component level and the *context-aware deployment plan* model which defines meta-information at the application level.

We use UML class diagrams for the graphical representation of the data model. As we do not have enough space, we do not give the graphical representation of all the model elements. We give examples of contracts written in XML progressively as we detail the PIM data model. These contracts result from the mapping of the PIM data model to the platform-specific model (PSM) for CCM. These examples are included in order to facilitate the comprehension of the model, but the mapping to PSM for CCM is detailed in Section 7.

5.1. Context model

In order to be able to adapt the deployment of an application to the context, component producers and application assemblers have to specify context elements that can impact the application deployment. This information allows the underlying context-aware middleware to identify the context sources to be used in order to collect the required contexts. Our model includes a description of context elements that can impact the application deployment. This description indicates for each context its identifier, its name and its type (see Fig. 4).

Component producers and application assemblers have also to specify the relevant context values that have particular impacts on the application deployment in order to be able to filter the collected context information. The data model of a relevant context description is shown in Fig. 5. A RelevantContext is associated to one of the contexts that impacts the deployment of the application. Each relevant context is described by a comparison operator ($>$, $<$, $=$, $\geqslant$ or $\leqslant$) and a particular context value. For example, if we consider the context representing the zone where a node is located, the relevant context "the node is in the primary medical zone" is described as in Fig. 6. It is possible to define a composition of relevant contexts collected from different sources.

We consider that the deployment service is offered by a service provider that provides a set of deployment nodes. The resources of these nodes and the network connections between them represent a context which is systematically taken into account for the deployment of all the applications. The description of this context is supported by the

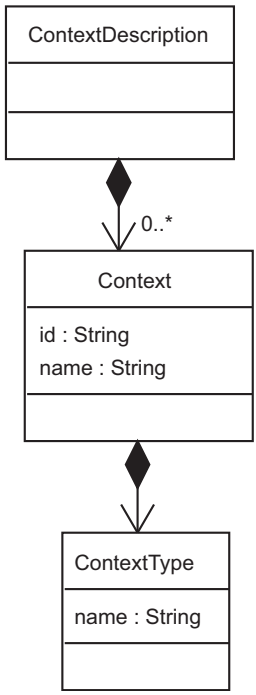

Fig. 4. Context description.

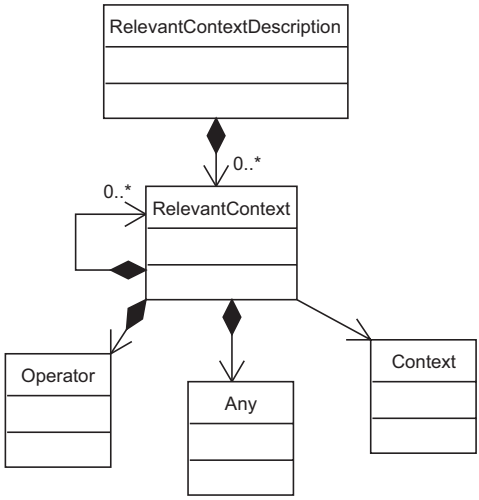

Fig. 5. Relevant context description.

```
<relevantcontext id= "PMnodeZone"  contextref="nodeZone">
  <operator  name="equal"/>
  <relevantvalue>PrimaryMedicalZone</relevantvalue>
</relevantcontext>
```

Fig. 6. Example of a simple relevant context description.

service provider. The deployment actors do not have to specify this context for each application.

5.2. Component package description model

The deployment adaptation meta-information situated at the component level is defined by a component package description model. This model has to support the classical deployment meta-information situated at the component level (see Section 3.2) and to extend it in order to support context-awareness.

Since this model is an extension of a traditional component package description, which does not support context-awareness, we chose to extend the component package description model of the D&C (OMG, 2006) specification. We made this choice because the D&C model is complete and platform independent (Ayed, 2005). Consequently, the component package model supports the description of the classical meta-information (see Fig. 7), such as: general meta-information (*GeneralDescription*) and structural meta-information (*ComponentInterfaceDescription*). In order to support context-aware deployment, our model also supports the description of the following meta-information: variability of the component properties according to the context (*CAPropertiesDescription*), description of the various implementations of the component (*ComponentImplementationDescription*) and the context placement constraints of each implementation

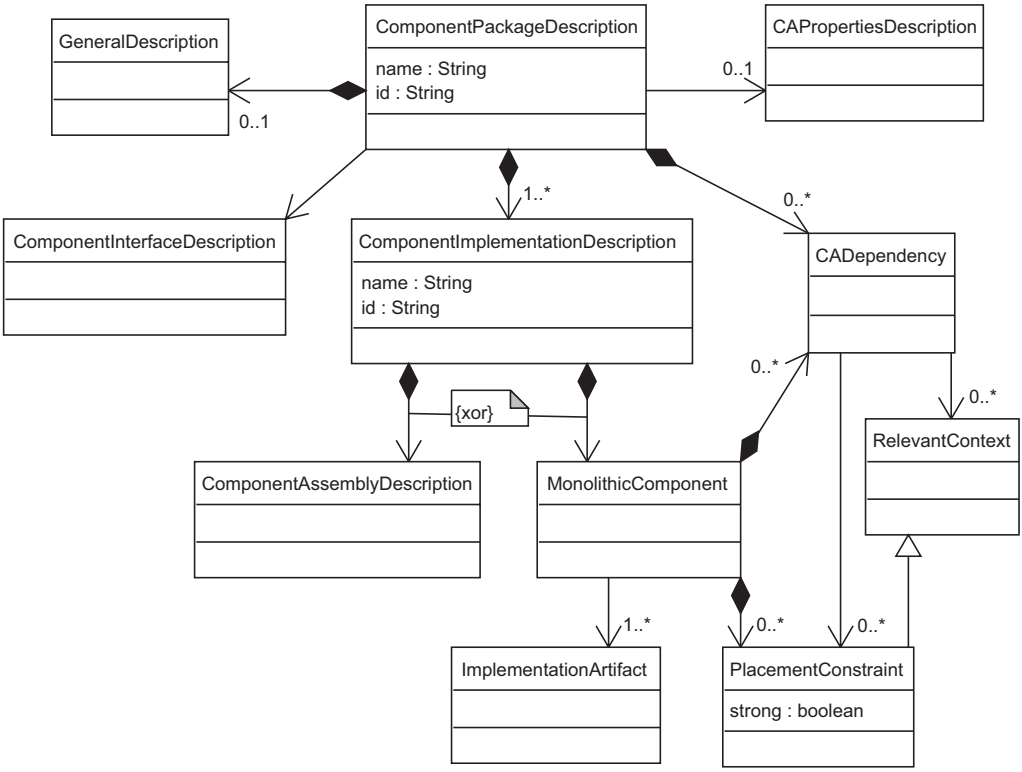

Fig. 7. Component package model.

(*PlacementConstraint*) and the variability of the dependencies of the component according to the context (*CADependency*).

A component can be simple or composite. The implementation of a simple component represents a compiled code (*MonolithicComponent*) and the implementation of a composite component represents an assembly of components (*ComponentAssemblyDescription*).

In the following we detail the various elements which extend a traditional component package description model: the properties variability, the placement constraints and the dependencies variability.

5.2.1. *Properties variability*

The component producer has two possibilities to specify the variability of the various functional and non-functional properties of a component. He can specify the various values that can be taken by each property by associating these properties to various relevant contexts (see Fig. 8, example Fig. 10), or he can directly associate a property to a particular context so that the value of the property takes the value of this context at the deployment time. This direct mapping of the properties values on the context is modeled by the *ContextMapping* element. The example of Fig. 9 shows a mapping of the configuration property "user language" on the context, which represents the user language.

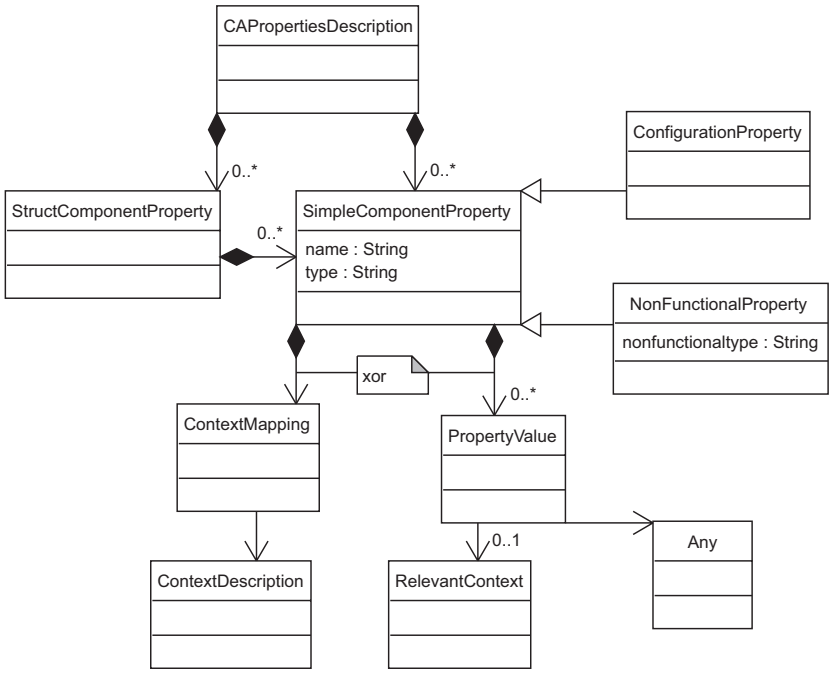

Fig. 8. Variation of properties values according to the context.

```
<capropertiesdescription>
     <simplecomponentproperty name="GUISupportedLanguage"     type="string">
         <contextmapping contextref="UserLanguage"/>
     </simplecomponentproperty>
</capropertiesdescription>
```

Fig. 9. Example of a component property mapping.

```
<simplecomponentproperty name="pers" type="string"   nonfunctionalkind="persistency" >
  <propertyvalue relevantcontextref="nonpersistencepreference"  value="transient"/>
  <propertyvalue relevantcontextref="persistencepreference"   value="stateful"/>
</simplecomponentproperty>
```

Fig. 10. Example of a non-functional property variability.

A non-functional property requires the specification of a non-functional type such as the security or the persistency types. Fig. 10 gives an example of a non-functional property that represents the persistency of the victim manager component. This property depends on the rescuer preference, i.e. whether he prefers to save the victims' conditions and diagnosis or not.

5.2.2. Placement constraints

The placement constraints allow the deployment process to determine the placement of a component and to choose one of its implementations. A component may have several implementations (see Fig. 7). Each one must have a description of the contexts in which it can run. The component producer specifies the context constraints required by each implementation of the component. There are two types of constraints: strong constraints and weak constraints. If the strong constraints of an implementation are not satisfied during the assignment of the component to a node, then this implementation cannot be selected for this assignment. The satisfaction of weak constraints is not compulsory, but only preferable (these constraints represent relevant contexts extended by a *strong* attribute). Fig. 11 shows the description of the placement constraints of one of the *view* component implementations. This implementation is dedicated to the rescuers who have a first-aid worker profile and can be installed only on a device that has the "Microsoft Pocket PC" operating system.

5.2.3. Dependencies variability

A component package may have dependencies on resources (see Section 3.2). These dependencies may vary according to the context, which explains the association of the *CADependency* element to zero or several relevant contexts (see Fig. 7). Each implementation of the component may have its own dependencies which are added to those of the component. For example, if the network connection bandwidth is weak during the deployment of the component *victim manager* on a node, then this component will have a dependency on a software which enables it to compress the data before sending them (see Fig. 12).

In this section we have modeled the meta-information that allows the deployment process to adapt the deployment parameters at the component level according to the

```
<componentpackagedescription name="viewcomponentpackage"   id="partnerpack">
  <componentimplementationdescription name="firstaidworkerimplem" id="rescimpl">
     <monolithiccomponent>
        <placementconstraint type=strong   relevantcontextref="PocketPCOS">
        <placementconstraint type= strong  relevantcontextref="firstaidworkerProfile">
      </monolithiccomponent>
  </componentimplementationdescription>
</componentpackagedescription>
```

Fig. 11. Example of a placement constraints description.

```
<componentpackagedescription name="VMngerpackage" id="serverpack">
   <cadependency  type="package" action="install"  relevantcontextref="lowbandwidth">
      <fileinarchive name="zippack.rar"/>
   </cadependency>
</componentpackagedescription>
```

Fig. 12. Example of a component dependency variability.

context: its properties, its dependencies, its implementation and its placement. We notice that components are developed and placed in packages to be sold and then integrated into applications that the component developer may ignore. Thus, the meta-information at the component level cannot take into account the semantics of the applications in which the component will be integrated. Consequently, some of them could be extended or overloaded at the application level in order to take into account the semantics of the application, as we will show in the next section.

5.3. Context-aware deployment plan model

A context-aware deployment plan represents a set of meta-information that makes it possible to plan the deployment of an application according to the context. Planning the deployment according to the context consists in specifying the variability of the deployment parameters at the application level (see Section 3.2). A context-aware deployment plan must consequently describe the variability of the application architecture according to the context. It may additionally define the variability of the parameters associated with each component instance. For this purpose, it either adds the meta-information that takes into account the semantics of the application to the meta-information defined at the component level or overloads it.

The context-aware deployment plan is represented by the data element *ContextAwareDeploymentPlan* (see Fig. 13). This element describes the set of the component instances (*CAComponentInstance*) that make up the application, as well as their connections (*CAConnectionDescription*) and their deployment parameters (placement and properties). The context-aware deployment plan is specified by the application assembler of the application. In the following we detail the elements of the context-aware deployment plan (see Fig. 13): the specification of the variability of the application architecture, the dynamic discovery of preinstantiated components, the variability of the placement of the component instances and the variability of the component instance properties.

5.3.1. Specification of the application architecture variability

One of the hardest tasks of the context-aware deployment is the modification of the application architecture according to the context because it requires the description of all its possible architectures for all the possible context states. The number of possible architectures can be large, which results in an exponentially growing number of combinatorial variants of the whole application. The description of all the possible architectures becomes very complicated in the case of large applications that have numerous components and are sensitive to many contexts.

To avoid repetitive descriptions, we propose that the application assembler should distinguish between two types of component instances when he specifies the architecture of an application: obligatory components, which are deployed for each context, and optional components, whose existence at the time of deployment depends on context and consequently requires the specification of an existence condition. In the *disaster management* application, the *GUI* component, the *victim manager* component and only one *view* component (that of the EOC) are obligatory components. The logging component and the remaining *view* components are optional components since their existence depends on the context.

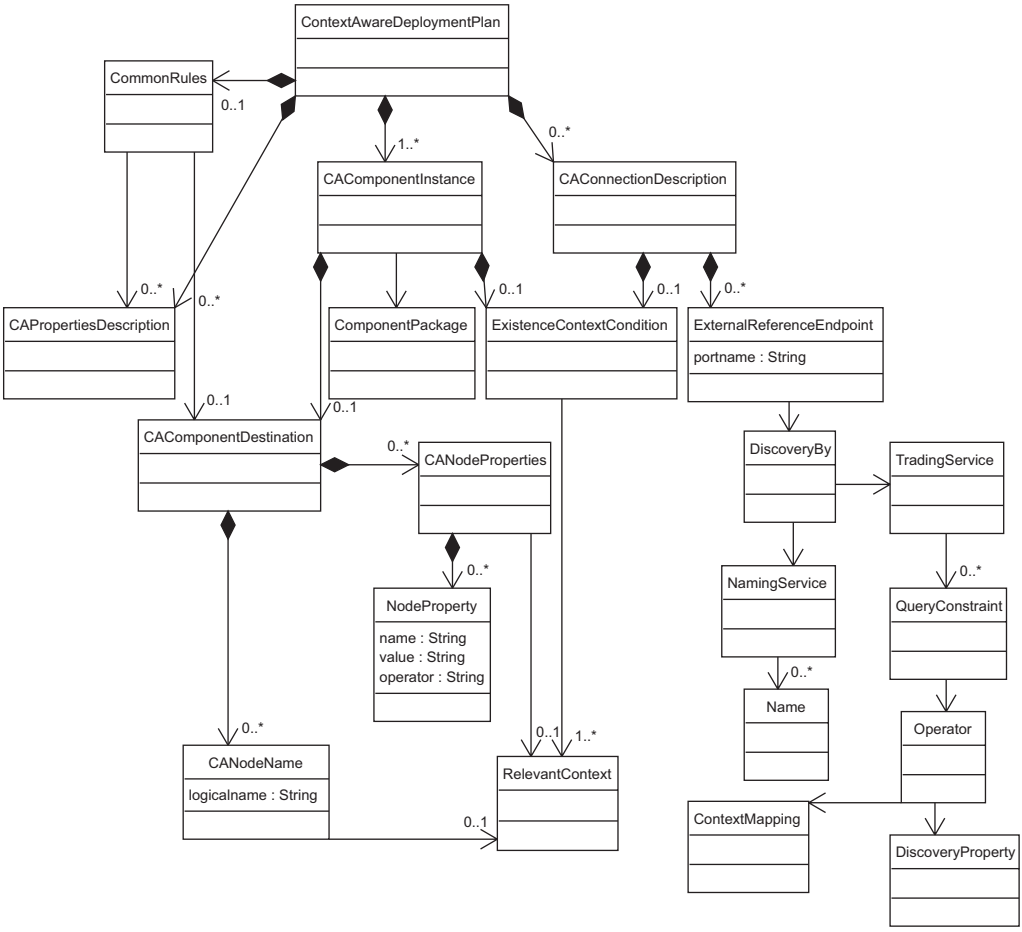

Fig. 13. Context-aware deployment plan.

We also distinguish between two types of connections: obligatory connections and optional connections. Optional connections also require the specification of a condition that determines their existence. 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 optional, but their existence condition is satisfied). The *ExistenceContextCondition* element (see Fig. 13) is expressed by a set of context constraints represented by a disjunction of relevant contexts.

Fig. 14 shows the specification of the optional connection between the *GUI* and the *logging* components. This connection will be deployed only if the user is located near a network uncovered zone or if the user prefers to work in a disconnected mode.

Besides the application architecture, the context-aware deployment plan supports the specification of dynamic discovery of preinstantiated components and, if necessary, the specification of the placement and the configuration of the properties of the components, by considering the application semantics.

```
<caconnectiondescription idref="GUI-LOG">
    <existencecontextcondition>
      <relevantcontextref idref="nearUncoveredZone"/>
      <relevantcontextref idref="disconnectedMode"/>
    </existencecontextcondition>
     <usesport>
       <usesidentifier>LOGinterfaceid</usesidentifier>
       <componentinstantiationref idref="GUIinstance"/>
     </usesport>
     <providesport>
       <providesidentifier>LOGinterfaceid</providesidentifier>
       <componentinstantiationref idref="logginginstance"/>
     </providesport>
</caconnectiondescription>
```

Fig. 14. Example of an optional connection description.

```
<ExternalReferenceEndpoint portname="DBinstExternalRef">
    <discoveryby><tradingservice>
        <queryconstraint idref="qconst">
          <operator value="equal">
            <discoveryproperty name ="DBzone"/>
            <contextmapping contextref="RescuerConnectionZone"/>
          </operator>
        </queryconstraint >
      <tradingservice></discoveryby>
</ExternalReferenceEndpoint>
```

Fig. 15. Example of a discovery request.

5.3.2. *Dynamic discovery of preinstantiated components*

The context-aware deployment plan allows the deployment planners to describe queries to be sent to a discovery service (Information technology-open distributed computing-ODP trading function, 1993) in order to find preinstantiated components and directly connect them to the application (*ExternalReferenceEndpoint* element of Fig. 13). Discovery queries may include context information (*TradingService* element in Fig. 13). For example, Fig. 15 specifies a discovery query in order to find a preinstantiated component representing the database of a hospital which is located in the same zone as the rescuer.

5.3.3. *Placement of the component instances*

The description of the placement constraints at the component level is generally sufficient to determine the placement of a component. However, in certain cases, the placement of a component instance requires the consideration of the application semantics in which it is integrated. In these cases, the specification of the placement of the component instance is carried out in the context-aware deployment plan of the application. The *logging* component is an example of a component for which the placement must be specified in the context-aware deployment plan. Indeed, the developer creates this

```
<cacomponentdestination>
    <canodename logicalname="UserTeminal  relevantcontextref="SufficientResources"/>
    <canodeproperties relevantcontextref="InsufficientResourcesAndConnected" >
            <nodeproperty name ="nodezone" operator ="equal"  value="userzone"/>
    </canodeproperties>
</cacomponentdestination>
```

Fig. 16. Example of a placement description.

component to be used by any component that needs the logging service. He specifies its general placement constraints at the component level so that it can be automatically placed on a node which satisfies these constraints. However, if we consider the semantics of the *disaster management* application, we know that the role of this component consists in logging the activities carried out by the rescuer during network disconnections. Therefore the logging component must be placed on the user device. Its automatic placement on any node satisfying only its general placement constraints does not make sense. Consequently, in the case of the disaster management application, the description of the *logging* component placement must be carried out in the context-aware deployment plan.

At the context-aware deployment plan, the placement of a component instance can be specified either by a logical name or by one or several nodes properties associated with relevant contexts (see *CAComponentDestination* in Fig. 13). These nodes are dynamically determined at deployment time through a discovery service. The example of Fig. 16 shows an excerpt of the *logging* component placement specification. This component will be placed on the user device when the latter has sufficient resources. If there is no risk of network disconnection, then this component will be placed on a node located in the same zone as the user.

When the placement of a component instance is specified in the context-aware deployment plan, the placement algorithm uses the placement constraints to look for an implementation of the component which can run on this node.

5.3.4. Variability of the component instance properties

The variability of the component properties can be specified at the component level, but by considering the application semantics, it is possible to discover other values that the attributes and the non-functional properties of the component instances can have. Just as at the component level, the variation of the properties according to the context is specified by the *CAPropertiesDescription* element (see Fig. 13). This element extends or overloads the values defined at the component level.

5.4. Conclusion

In this section, we have defined the model of the meta-information which makes it possible to adapt the deployment parameters to the context. This meta-information allows the deployment process to vary the deployment parameters related to a component (its properties, its dependencies, its implementation and its placement) as well as the deployment parameters related to the application (the general architecture of the application and the placement of its component instances and their properties). In Section 7, we describe the mapping of this data model to concrete deployment contracts.

6. CADeComp execution model

Contrary to the data model, which is a model of descriptive information, the execution model specifies the execution entities that process, create or provide these data. In this section we first give an overview of these entities, and then we describe the deployment adaptation process carried out by these entities.

6.1. Execution entities overview

We suppose that the deployment service is offered by a deployment provider. The CADeComp architecture is presented in Fig. 17. The deployment service provider offers a set of nodes called execution servers which provide an execution environment for component instances. These nodes are utilized when the user device is overloaded and is unable to host some of the components of the application. All of these nodes have a component middleware and a context-aware middleware. These nodes represent the deployment domain.

The *DeploymentExecutor* entity represents a classical deployment tool provided by the component-based middleware. It is able to perform the deployment activities defined in Section 3.1.

The deployment service provider has two repositories: the component package repository and the application meta-data repository. The *component package repository* contains the component packages of the applications to be deployed. A component package contains various implementations of the component and meta-information necessary for the deployment adaptation of the component (see Sections 5.1 and 5.2). The *application meta-data repository* contains the meta-information necessary to adapt the deployment of the applications offered by the service provider (see Sections 5.1 and 5.3).

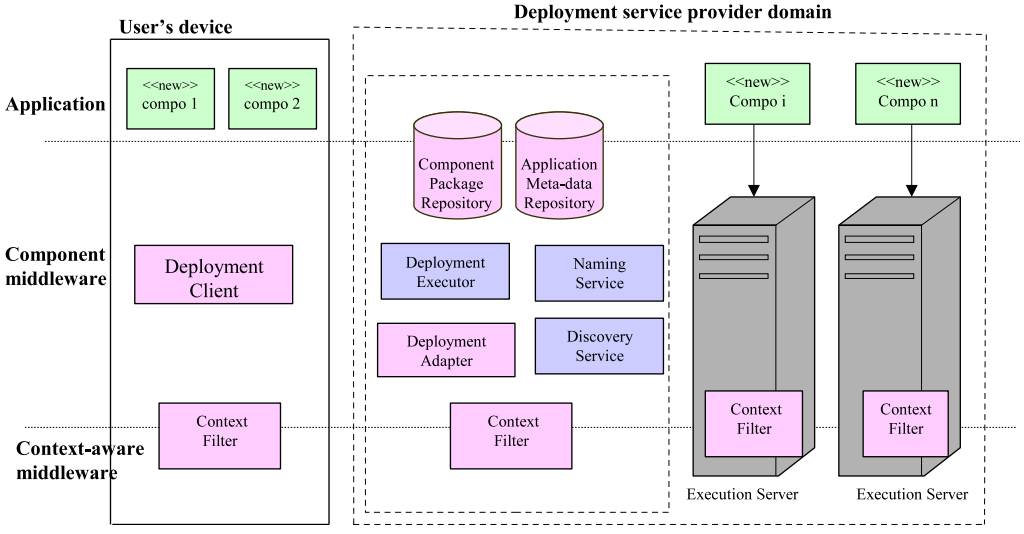

Fig. 17. CADeComp architecture.

To be able to deploy an application, the user device needs a component-based middleware, a context-aware middleware and a *DeploymentClient*. The *DeploymentClient* is an entity which allows a user to initiate the deployment of an application. It sends a deployment request to the *DeploymentAdapter* that includes the name of the application to be deployed and the user preferences.

The adaptation of the deployment to the context in CADeComp is carried out by two main entities: the *ContextFilter* and the *DeploymentAdapter*.

The *ContextFilter* filters the relevant contexts for the deployment of a given application. Thanks to it, the network bandwidth is saved because the context information is transferred into the network only if it is relevant for the application deployment. Since context information must be collected and filtered from all the nodes of the deployment domain, each node must run a *ContextFilter*.

The *DeploymentAdapter* is responsible for all the deployment adaptation decisions. When it receives a deployment request from a *DeploymentClient* for a given application, the *DeploymentAdapter* recovers the meta-information associated with this application from the application meta-data repository. Then it uses the context-aware deployment plan in order to identify the component packages of the component package repository that will be used in the deployment of the application. The *DeploymentAdapter* sends to the *ContextFilters* of the user device and the deployment domain the description of the relevant contexts to be filtered. Each *ContextFilter* interacts with the context-aware middleware in order to filter the relevant context information for the deployment of the application.

The *DeploymentAdapter* analyzes the collected relevant contexts in order to take deployment adaptation decisions and produces the final deployment plan of the application (see Section 3.2). The final deployment plan specifies the application architecture that will finally be deployed. For each component of the application, it specifies a placement node, an implementation to be used for the component instantiation, a set of dependencies and fixed values for its various properties. The *DeploymentAdapter* sends the generated final deployment plan to the *DeploymentExecutor* which deploys the application.

The *DeploymentAdapter* takes adaptation decisions in accordance with the adaptation meta-information situated at the component level and application levels through the deployment adaptation algorithm. The CADeComp deployment adaptation algorithm was detailed in (Ayed et al., 2006). In the following section, we describe briefly the stages carried out by the *DeploymentAdapter*.

6.2. Deployment adaptation process

After collecting the relevant contexts from the context filters, for each component instance described at the application level, the *DeploymentAdapter* checks its existence; if the component is obligatory or if it is optional and its existence condition is satisfied by context then the component is put in the final deployment plan. But if not, then it will not be deployed. In the same manner, the *DeploymentAdapter* checks the existence of the connections specified at the application level in order to determine the connections between the components in the final deployment plan.

Once the architecture of the application is determined, the *DeploymentAdapter* determines the placement of the components. The components whose placement rules

have been specified at the application level are assigned to nodes by applying these rules. Concerning the other component instances, the *DeploymentAdapter* extracts the placement constraints situated at the component level of each component and finds the possible placement nodes for each of its implementations, i.e. the nodes that satisfy their strong constraints. As there can be several valid assignments for the same component, the *DeploymentAdapter* selects the best one. The best assignment can be determined according to several different criteria, such as maximizing the number of satisfied weak constraints or optimizing the resources consumption. We use the A* (Nilson, 1980) algorithm in order to explore all the possible assignments and to approach the best one. More details about the component placement algorithm, the optimization of the resources consumption and the calculation of the number of satisfied weak constraints are given in (Beloued et al., 2005).

In order to assign the components' properties values (functional and non-functional), the *DeploymentAdapter* extracts the application level rules related to the properties and applies them. Then it extracts the component level rules of the components whose properties variation has not been specified at application level. If for a given property none of its associated rules can be applied because their constraints are not satisfied by context, then the property will have its default value.

After these stages are carried out, a deployment plan can be generated. Then the classical deployment tool (the *DeploymentExecutor*) performs the deployment.

7. PSM and experimentation on CCM

In Section 5 and 6 we have presented a PIM for CADeComp. In order to be able to use CADeComp on a particular platform, we have to specify a PSM for CADeComp.

We have chosen to place CADeComp above a CCM component-based middleware (OMG, 2002) because it has the most complete deployment tool among the deployment tools that we have studied. The D&C deployment specification is more complete, but it has not been implemented yet. In this section we define a CADeComp PSM for CCM. Then we discuss some experimental results obtained after the implementation of CADeComp on OpenCCM (OpenCCM) in order to finally evaluate CADeComp.

7.1. CADeComp PSM for CCM

The CADeComp PSM for CCM is defined through three transformations:

- The first transformation, called PIM to PIM for CCM, takes the PIM and refines it into a PIM for CCM. In this PIM for CCM, the abstract meta-concepts are concretized and aligned with the CORBA component model.
- The second transformation, called PIM for CCM to PSM for CCM for XML, creates a PSM for CCM for XML that can be used to generate contracts written in XML.
- The third transformation, called PIM for CCM to PSM for IDL, takes the PIM for CCM and transforms it into a PSM for CCM for IDL that can be used to generate concrete IDL from the model.

In the following we briefly describe these transformations.

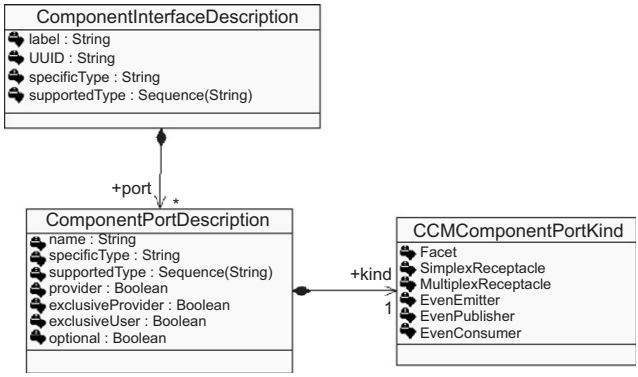

Fig. 18. Description of a CCM component interface.

7.1.1. PIM to PIM for CCM transformation

This transformation takes the PIM and aligns classes with the CORBA component model. This involves changes to attributes and semantics of some classes. All classes from the PIM that are not refined here are imported into the PIM for CCM without change.

The current CCM model does not support the composite components. Thus the implementation of a component can only be monolithic. This leads us to omit the *ComponentAssemblyDescription* class from the component package description model (see Fig. 7).

The *ComponentInterfaceDescription* and *ComponentPortDescription* classes are augmented to support CCM (see Fig. 18). The idlFile attribute is added to the *ComponentInterfaceDescription*. It contains a URL that references an IDL file containing the definition of this component. The kind attribute added to the ComponentPortDescription specifies the port kind that is used (facet, receptacle, event source, event sink, etc.) (OMG, 2006).

To align the final deployment plan with the descriptors defined in CCM, the DeploymentAdapter generates a CCM assembly descriptor. This descriptor does indeed contains the various elements specified in the final deployment plan.

7.1.2. PIM for CCM to PSM for CCM for XML transformation

The CADeComp data model elements are used to generate XML DTDs which will be used to define the CADeComp contracts. The main classes of the data model (ContextDescription, RelevantContextDescription, ComponentPackageDescription and CADeploymentPlan) are mapped to XML elements files. The contained associations of the main classes will be mapped into XML elements within these files. We use the following transformation rules in order to obtain the XML files from the CADeComp data model:

- A class is mapped into an XML element.
- Composition associations imply that the class at the part end is in the same XML file as the class at the composite (containing) end. The class at the composite end will be mapped into a father XML element and the class at the part end will be mapped into a child XML element.
- Non-composite associations between two classes imply that they belong to two different files. These associations are implemented by an identifier attribute at the target and by a matching reference attribute at the source.

- Inheritance relationships are removed; all attributes and associations of the base class are attached to the derived class.

The XML files obtained after the application of the transformation are the following:[2] a *context descriptor*, a *relevant context descriptor*, a *context-aware component contract* and a *context-aware deployment plan contract*.

7.1.3. PIM for CCM to PSM for CCM for IDL transformation

Execution model elements are mapped to CORBA IDL. The mapping to IDL is accomplished using the rules set forth in the UML profile for CORBA. To apply these rules, the classes defined in the execution model are mapped to the CORBAInterface stereotype so that these classes become CORBA interfaces.

7.1.4. Conclusion

The MDA approach that we followed in CADeComp makes it possible to extend several deployment tools. The definition of a PIM allowed us to specify transformations in order to automatically generate a PSM for CCM. This approach allows us to reuse the same model for several other component middleware and context-aware middleware and to plug CADeComp into several types of deployment tools.

7.2. Implementation and evaluation of CADeComp

We argue that context-aware deployment presents an important added value for mobile users. However, this context-awareness should not add significant latencies compared to classical deployment. It thus seemed essential to us to make an experimental study of CADeComp to evaluate deployment times. In this subsection, we first introduce the implementation and the evaluation platform. Then we discuss the validation tests and the deployment time measurements. Finally we evaluate CADeComp.

7.2.1. Demonstration platform and applications

We implemented CADeComp above OpenCCM, an implementation of the CCM specification. In this implementation, the DeploymentAdapter invokes the CCM deployer after generation of a CORBA component assembly and other CCM descriptors needed by OpenCCM deployer, such as component file descriptors and CORBA component descriptors. The ContextFilters use a light context-aware middleware that we developed. This middleware uses context acquisition operations. The signature of these operations is given in context descriptors. Deployment clients invoke CADeComp with CORBA requests.

In order to demonstrate and validate CADeComp, we developed several applications: the disaster management application, an online shopping application which was described in (Ayed, 2005), and other applications of demonstrations that are in the CADeComp web site.[3] The tests and results that we will show in the following are based on the disaster management application.

During the deployment tests, the `DeploymentAdapter` ran on Red Hat Linux 9.0 powered by a 1.8 GHz Pentium Intel with 256 MB of RAM. Besides this node we have four

[2]The DTDs of these descriptors are on the CADeComp web site http://picolibre.int-evry.fr/cgi-bin/cvsweb/cadecomp/src/.

[3]http://picolibre.int-evry.fr/cgi-bin/cvsweb/cadecomp/src/.

other nodes which have the same capacities and which play the role of execution servers. All of them are connected via an Ethernet network. The PDAs used are iPAQ with Microsoft Pocket PC 4.20.0 powered by a TI OMAP1510 processor and 56 MB of memory and having a WiFi connection.

7.2.2. Validation tests

We made a validation test of CADeComp for the deployment of the disaster management application. For that purpose, we considered the different context types that impact the deployment of the partners application, which are as follows: the rescuers' profiles (which impact implementation choices), the number of rescuers (which impacts the number of deployed components), the rescuers preferences (which impact component configuration properties and the application architecture), the location of the rescuer (which impacts component configuration) and the resources provided by the rescuer device, as well as the execution servers load (which impacts component placement and the implementation choices). We have tested the deployment of the application with five different rescuers profiles and different preferences of the rescuers concerning their languages and the displaying colors. We varied the number of rescuers from 2 to 51. The location of the rescuers was simulated by an application which gives randomly the zone where the rescuer is located. We have varied the resources and the load of the nodes as well as the network connection bandwidth between them. These validation tests showed that the applications deployed by CADeComp are in accordance with what the user would expect in terms of context adaptation.

7.2.3. Measurements of the deployment time

Fig. 19 shows the average deployment time of the disaster management application, with and without adaptation according to the number of deployed components (we vary the

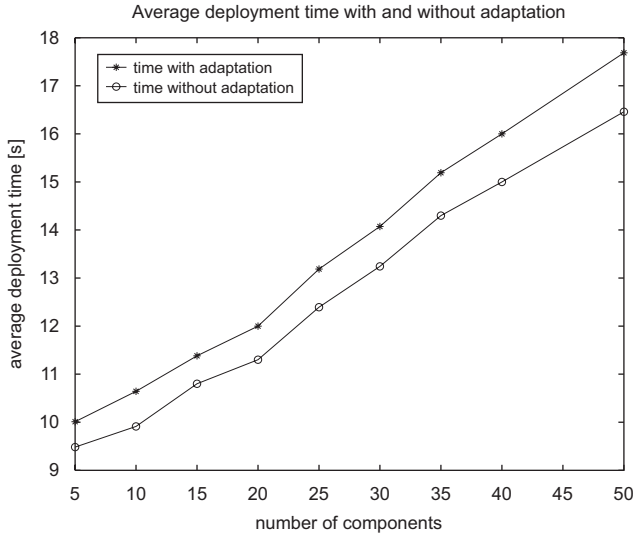

Fig. 19. Average deployment time of the disaster management application according to the number of deployed components.

Table 1
Structure of the disaster management application

Component types	GUI	Victim manager	View	Logging
Number of implementations	5	1	5	1
Average package size (kB)	120	130	125	100
Number	1	1	Depends on the number of partners	0 or 1
Placement	Rescuer device	One of the execution servers	Partner device	Rescuer device

number of deployed *view* components). Details about the disaster management application are given in Table 1. Each test was executed 20 times in order to obtain meaningful averages. The average variance of these measurements is 0.76. The first curve (without adaptation) represents the time taken by the CCM deployer to deploy the application (components downloading, installation, configuration and activation). The second curve (with adaptation) represents the CADeComp deployment time.

Let $D(n) = Adapt(n) + Depl(n)$ with n, the number of deployed components, $D(n)$, the total deployment time of the application, $Adapt(n)$, the deployment adaptation time and $Depl(n)$, the deployment time without adaptation.

The results show that between 5 and 50 components, the adaptation time $Adapt(n)$ increases linearly with the number of components at an average of 0.21%. Since we have a number of adaptation rules associated with each component to deploy at the application and component levels (an average of 20 rules per component for the disaster management application), the number of rules increases with the number of components in the application. This explains the delay increase corresponding to the increasing number of components. $Adapt(n)$ represents an average of 5.48% of $D(n)$.

$Adapt(n)$ represents the processing time of the adaptation rules and the amount of time necessary for preprocessing the rules and generating the final deployment plan (cf. Fig. 20). Preprocessing the rules consists in reading the files containing the rules at the application and component levels (at the component level we have one file of rules for each component).

$Adapt(n) = RulesPreprocessing(n) + RulesProcessing(n) + DeploymentPlanGeneration$

$RulesProcessing(n)$ represents the processing time of the architecture rules, the processing time of the properties rules and the processing time of the placement rules:

$RulesProcessing(n) = Arch(n) + Conf(n) + Place(n)$.

Fig. 20 shows the average time necessary for the processing of the architecture rules, the properties configuration rules and the placement rules as well as the total adaptation time of the disaster management application, according to the number of deployed components. The processing time of the properties configuration rules is the greatest because each component has several properties and, for OpenCCM deployer, we have to generate a property file for each component after determining the values of its properties. These delays could be reduced if we directly transmit the parameters to the deployment tool without utilizing files in between. The processing time necessary for the placement rules is low because the placement of the user interface component and the entire view components

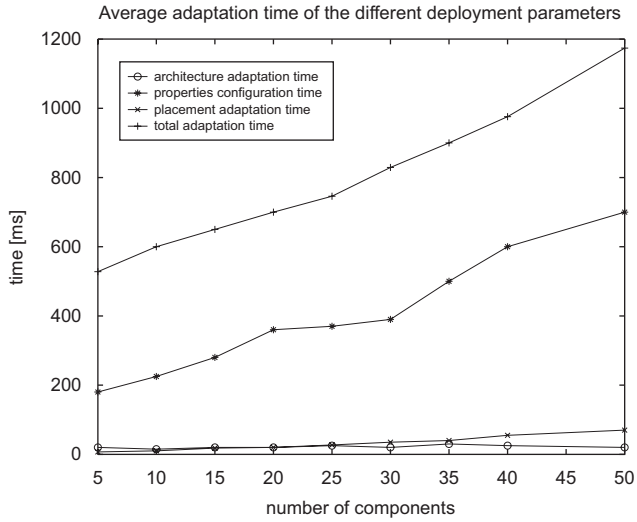

Fig. 20. Average adaptation time of the different deployment parameters of the partners' application.

was specified at the level of the application rules (since it is known that they have to be placed on the rescuers' devices). Only the server component is automatically placed by the placement algorithm.

7.2.4. Usability and applicability of CADeComp

CADeComp deployment is processed just-in-time (the components are downloaded, instantiated and activated at application access time) and automatically in order to relieve users of the manual and repetitive deployment tasks. Applications are always up-to-date (the latest version is always downloaded) and the memory may be freed after each application execution. Just-in-time deployment assumes the existence of a networking infrastructure from which the software should be installed. Our hypothesis is that this infrastructure is offered by a service provider. For example, in the case of the large-scale disaster scenario, this infrastructure is located in one of the ambulances driven to the disaster area. We assume that the user is able to connect to this infrastructure at the deployment time. This connection does not necessarily need to be maintained active during all the execution of the application, but it is necessary for the deployment of the application.

CADeComp deploys an application for a given use and a given context. As current CADeComp implementation does not support deployment reconfiguration during execution, this supposes either that the context state collected at the deployment time remains stable during the execution of the application, or that despite the changes the deployment parameters calculated by CADeComp are still acceptable. For example, the disaster management application is deployed for a given disaster in a given area. Contexts such as rescuers profiles and the area do not change during the rescue process. We made the hypothesis that the placement remains acceptable for the entire application execution, even if the nodes' loads change slightly.

7.3. Advantages and shortcomings of the CADeComp model

The CADeComp data model and execution model allow a classical deployment process to be context-aware. The advantage of CADeComp compared to existing solutions is that it is not limited to a special type of context such as the user preferences or the hardware resources (see Section 2.2). It is able to consider various types of context. In addition, the CADeComp model is not dedicated to a specific platform: it is platform independent, which allows us to map it to several platforms.

Thanks to its two-level structure (component level and application level) directed by the different types of deployment parameters, the data model provides component producers and application assemblers with more independence from one another and more flexibility. It offers them simple means to organize the specification of the deployment variability in order to be able to cover the major part of the possible variations. The possibility of description of discovery requests ensures a dynamism in the deployment. The advantage of using the contracts is their simplicity but their disadvantage is that the component producers and the application assemblers have to imagine and describe all the possible variations and their associated context constraints.

The component producers or the application assemblers can introduce some inconsistencies when writing the contracts on the basis of this data model. For instance, the architecture variability description may lead to inconsistencies which could cause the deployment of one or several detached components or connections. A detached component is deployed without the connections that connect it to the remaining components of the application. A detached connection is deployed without components to connect to it. In Ayed et al. (2006) we defined a formal structure of the CADeComp data model and we analyzed the different types of inconsistencies that can be introduced when the adaptation rules are described. In (Ayed, 2005) we developed a checking tool that detects inconsistencies and proposes possible corrections to the application assembler.

8. Conclusion

In this paper we propose a middleware entity entitled CADeComp to adapt the deployment of component-based applications to the context. CADeComp is based on an approach which takes into account the context information during the various phases of component production and assembly to build an application. Taking into account the context information consists in describing a set of meta-information which is thereafter interpreted by entities which carry out the adaptation of the deployment to the context.

We defined a platform-independent data model in order to be able to describe the meta-information used to adapt the deployment to the context. This meta-information is situated at the component and application levels and is related to the semantics of the application. It describes the context which impacts the deployment of the application as well as contracts which specify the variation of the six deployment parameters according to this context. Thanks to its two-level structure, which is directed by the different types of deployment parameters, the data model provides component producers and application assemblers with more independence from one another, more flexibility and more intuitivism. It is based on rules that are easy to specify, and it offers the possibility to describe discovery requests which ensure dynamism in the deployment.

We defined an architecture of the deployment adaptation of component-based applications. This architecture is based on a set of platform-independent entities which can be added above a classical deployment tool. These entities collect context information from a context-aware middleware and use the adaptation meta-information to make deployment decisions.

The originality of CADeComp is that it is not dedicated to a specific platform: we can map it to a number of platforms. We mapped the data and the execution models of CADeComp on the CCM component-based middleware. The mapping of the data model results in a set of deployment contracts at the component and application levels. The mapping of the execution model results in a set of IDL interfaces which carry out the adaptation of the deployment to the context.

We implemented CADeComp on OpenCCM and we evaluated the CADeComp deployment time by comparing it to the deployment time without context adaptation. We found that the delays added by the deployment adaptation and automation are negligible compared to their added value.

CADeComp satisfies each of the four requirements that we mentioned in the Introduction: deployment automation, deployment adaptation, just-in-time deployment and negligible delays. The advantage of CADeComp compared to existing solutions is that it is based on a generic adaptation approach which can take into account several types of contexts. It is not limited to a special type of context.

We mapped our deployment model on CCM because it provides the richest deployment tool. In further phases we will map our model to other platforms, such as .NET or J2EE.

Even if the deployment contracts are easy to write, it remains difficult for the deployment actors to imagine and to describe all the possible variations. To avoid this inconvenience, we want to extend our data model by an ontology that will allow us to use an inference engine. This inference engine will automatically infer adaptation rules from basic ones specified by the deployment actor.

The methodology specified in this paper adapts the initial deployment of applications to the context (installation, instantiation and activation of components). In further phases, we intend to extend this methodology to support the dynamic reconfiguration of deployed applications in order to be able to vary the deployment parameters during the execution of the application (add, delete or modify a configuration property of a component). 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. It requires the definition of other adaptation entities that will guaranty the components' interactions consistency during the reconfiguration as well as the extension of the data model in order to describe the possible reconfigurations after a context change. In Ayed et al. (2005) we proposed an initial solution for the support of reconfiguration in CADeComp.

Our solution is based on just-in-time deployment, which leads to the uninstallation 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 constantly changing. But if the users frequently need to use the same application, another solution can consist in using different policies to uninstall the deployed applications only when we need resources, such as the LRU or the FIFO policies.

References

Ayed D. Context-aware deployment of component-based applications. PhD thesis, National Institute of Telecommunications, Paris, November 2005 [in French].

Ayed D, Taconet C, Bernard G. Context-aware deployment of multi-component applications. In: The fifth generative programming and component engineering (GPCE) young researchers workshop 2003, Erfurt, Germany; 2003.

Ayed D, Belhanafi N, Taconet C, Bernard G. Deployment of component-based applications on top of a context-aware middleware. In: The international conference on software engineering, Innsbruck, Austria; February 2005.

Ayed D, Taconet C, Bernard G, Berbers Y. An adaptation methodology for the deployment of mobile component-based applications. In: IEEE international conference on pervasive services, Lyon, France; June 2006.

Beloued A, Taconet C, Ayed D, Bernard G. Automatic placement of components during the deployment of component-based applications. In: Actes des Journes Composants (JC 05), Le Croisic, France; 2005 [in French].

Blair GS, Coulson G, Robin P, Papathomas M. An architecture for next generation middleware. In: Proceedings of the IFIP international conference on distributed systems platforms and open distributed processing, London; 1998.

Bobrow D, Gabriel R, White J. CLOS in context—the shape of the design space. In: Object-oriented programming: the CLOS perspective. Cambridge, MA: MIT Press; 1993.

Capra L, Emmerich W, Mascolo C. CARISMA: context-Aware Reflective Middleware System for Mobile Applications. IEEE Transactions on Software Engineering 2003;29(10):929–45.

Chen H. An intelligent broker architecture for pervasive context-aware systems. PhD Thesis, University of Maryland, Baltimore County, 2004.

Chen H, Finin T, Joshi A. An ontology for a context aware pervasive computing environment. In: The proceedings of the IJCAI workshop on ontologies and distributed systems (IJCAI 03), Acapulco, MX; August 2003.

David P, Ledoux T. An infrastructure for adaptable middleware. In: DOA'02. Irvine, CA, USA: Springer; October 2002.

Dey A. Understanding and using context. Personal and Ubiquitous Computing Journal 2001;5(1):4–7.

Dey A, Abowd G, Salber D. A conceptual framework and toolkit for supporting the rapid prototyping of context-aware applications. Human–Computer Interaction 2001;16(2–4):97–166 [special issue on context-aware computing].

Dowling J, Cahill V. The K-component architecture meta-model for self-adaptive software. In: Reflection 2001; 2001.

Fahy P, Clarke S. A middleware for mobile context-aware applications. In: Workshop on context awareness MobiSys 2004; 2004.

Grace P, Blair GS, Samuel S. Remmoc: a reflective middleware to support mobile client interoperability. In: International symposium on distributed objects and applications (DOA), Catania, Sicily, Italy; November 2003.

Gu T, Pung HK, Zhang DQ. A middleware for building context-aware mobile services. In: IEEE vehicular technology conference (VTC), Milan, Italy; 2004.

Hall RS, Heimbeigner D, Wolf AL. A cooperative approach to support software deployment using the software dock. In: International conference on software engineering; 1999. p. 174–83 URL: ⟨citeseer.nj.nec.com/article/hall99cooperative.html⟩..

Information technology-open distributed computing-ODP trading function. ISO/IEC JTC1/SC21.59 draft, ITU-TS-SG 7 Q16 Report, November 1993.

IST, COACH WP2: specification of the deployment and configuration, ⟨www.ist-coach.org⟩, D2.4, July 2003. URL: ⟨www.ist-coach.org⟩.

Ivan A-A, Harman J, Allen M, Karamcheti V. Partitionable services: a framework for seamlessly adapting distributed applications to heterogeneous environments. In: The 11th IEEE international symposium on high performance distributed computing, Edinburgh, Scotland, UK; 2002. p. 103–12.

Jena2. ⟨http://jena.sourceforge.net/⟩.

Kiczales G. Aspect-oriented programming. Surveys 28A(4).

Meyer B. Applying design by contract. In: IEEE computer; 1992. p. 40–51.

.Net framework. ⟨http://msdn.microsoft.com/netframework/⟩.

Nilson NJ. Principles of artificial intelligence. San Francisco: Morgan Kaufmann; 1980.
OMG. CORBA components version 3.0: an adopted specification of the object management group. OMG documents formal/02-06-65, June 2002.
OMG. MDA guide version 1.0.1. omg/2003-06-1, June 2003.
OMG. Deployment and configuration of component based distributed applications version 4.0. OMG documents formal/06-04-02, April 2006.
Open Services Gateway Initiative. OSGi services platform specification, Release3, March 2003. URL: ⟨http://www.osgi.org⟩.
OpenCCM. ⟨http://openccm.objectweb.org⟩.
Sabri N. A CORBA component based architecture for personalized services. PhD thesis, University of Evry Val d'Essonne, June 2003 [in French].
Sorensen C-F, Wu M, Sivaharan T, Blair GS, Okanda P, Friday A, et al. A context-aware middleware for applications in mobile ad hoc environments. In: Middleware for pervasive and ad-hoc computing; 2004. p. 107–10.
Sun Microsystems, Enterprise JavaBeans specification 2.0, 2002.
Sun Microsystems. Java 2 platform, enterprise edition specification version 1.4, 2003. URL: ⟨ http://java.sun.com/j2ee/docs.html⟩.
Szyperski C. Component software, beyond object-oriented programming, 2nd ed. Reading, MA: Addison-Wesley; 2002.