Continuity of user tasks execution in pervasive environments

Imen Ben Lahmar

DOCTORAT EN CO-ACCREDITATION
TELECOM SUDPARIS ET L'UNIVERSITE EVRY VAL D'ESSONNE

Spécialité : Informatique

Ecole doctorale : Sciences et Ingénierie

Présentée par

Imen BEN LAHMAR

Pour obtenir le grade de
DOCTEUR DE TELECOM SUDPARIS

Continuity of User Tasks Execution in Pervasive Environments

Soutenue le 15 Novembre 2012 devant le jury composé de :

Rapporteurs

M. Noel De Palma	Professeur, Université de Grenoble
M. Khalil Drira	Directeur de recherche CNRS, Université de Toulouse

Examinateurs

M. Jean-Paul Arcangeli	Maître de conférences, HDR, Université Paul Sabatier
Mme Françoise Baude	Professeur, Université Nice Sophia-Antipolis
M. Djamel Belaïd	Directeur d'études, Télécom SudParis
M. Guy Bernard	Professeur, Télécom SudParis

Thèse n° 2012TELE0028

Acknowledgements

First, I would like to express my deepest sense of gratitude to my supervisor Dr. Djamel Belaïd for his patient guidance, encouragement and excellent advice throughout this study. His constructive suggestions were a source of inspiration and motivation all along my way. I would like also to express my sincere gratitude to my thesis advisor Pr. Guy Bernard for his direction and significant feedback.

I would like to thank all members of the jury for their attention and thoughtful Comments.

I owe gratitude to the members of the computer science department of Telecom Sud-Paris and to my fellows whose friendly company, during my thesis work, was a source of great pleasure.

Finally, I want to dedicate this thesis to my parents, Ali and Halima, who have given me the chance of a good education, and so much love and support over the years. I probably owe them much more then they think. I am deeply and forever indebted to them for their love, support and encouragement. I am very grateful to my brothers Karim and Houssem who has always been a source of encouragement. I am also very grateful and indebted to my husband Youssef and to my son Mohamed-Ayoub for their patience, love and encouragement.

Abstract

The proliferation of small devices and the advancements in various technologies have introduced the concept of pervasive environments. In these environments, user tasks can be executed by using the deployed components provided by devices with different capabilities. One appropriate paradigm for building user tasks for pervasive environments is Service-Oriented Architecture (SOA). Using SOA, user tasks are represented as an assembly of abstract components (i.e., services) without specifying their implementations, thus they should be resolved into concrete components.

The task resolution involves automatic matching and selection of components across various devices. For this purpose, we present an approach that allows for each service of a user task, the selection of the best device and component by considering the user preferences, devices capabilities, services requirements and components preferences.

Due to the dynamicity of pervasive environments, we are interested in the continuity of execution of user tasks. Therefore, we present an approach that allows components to monitor locally or remotely the changes of properties, which depend on. We also considered the adaptation of user tasks to cope with the dynamicity of pervasive environments. To overcome captured failures, the adaptation is carried out by a partial reselection of devices and components. However, in case of mismatching between an abstract user task and a concrete level, we propose a structural adaptation approach by injecting some defined adaptation patterns, which exhibit an extra-functional behavior.

We also propose an architectural design of a middleware allowing the task's resolution, monitoring of the environment and the task adaptation. We provide implementation details of the middleware's components along with evaluation results.

Résumé

L'émergence des technologies sans fil et l'ubiquité des dispositifs mobiles ont introduit le concept des environnements pervasifs. Dans ces environnements, les tâches d'un utilisateur peuvent être exécutées en utilisant des composants déployés sur des dispositifs ayant des capacités différentes. Un paradigme approprié pour la construction de ces tâches est le Service-Oriented Architecture (SOA). En utilisant l'architecture SOA, les tâches d'un utilisateur sont représentées par un assemblage de composants abstraits (les services), sans préciser leurs implémentations, d'où la nécessité de résoudre les services en composants concrets.

La résolution d'une tâche implique la sélection automatique des composants concrets à travers différents dispositifs de l'environnement d'exécution. Pour ceci, nous présentons une approche qui permet à chaque service d'une tâche de l'utilisateur, la sélection du meilleur dispositif et composant en tenant compte des préférences de l'utilisateur, des capacités des dispositifs, des besoins des services et des préférences des composants.

En raison de la dynamicité des environnements pervasifs, nous nous sommes intéressés aussi à la continuité d'exécution des tâches de l'utilisateur dans ces environnements. Pour cet objectif, nous présentons une approche qui permet aux composants de surveiller localement ou à distance les changements de propriétés fournies par d'autres composants. Nous avons également considéré l'adaptation des tâches de l'utilisateur en proposant une première approche de re-sélection partielle de dispositifs et de composants. Nous proposons aussi une approche d'adaptation structurelle par l'injection des patrons d'adaptation, qui offrent un comportement extra-fonctionnel.

Nous avons conçu l'architecture d'un middleware permettant la résolution des tâches, le monitoring de l'environnement et l'adaptation des tâches. Nous donnons quelques éléments d'implémentation des composants du middleware et nous présentons des résultats d'évaluation.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Thesis Context

Driven by the widespread of communication technologies and mobile devices, there is a growing trend in now-a-days for pervasive computing that allow users to access to any information using any device and any network at any time (Weiser, 1991). In such environments computing is pushed away from the traditional desktop to small hand-held and networked computing devices that are present everywhere. This creates an environment that hides the complexity of new technologies and enables users to benefit from their provided functionalities anywhere and at any time.

Moreover, software applications have evolved from centralized and stable applications, to highly decentralized and dynamic ones. This evolution has induced a change in the development of applications. One change come in the form of task-based computing (Sousa and Garlan, 2002) (Ben Mokhtar et al., 2007) which enables users to explicitly define their tasks that are then realized using networking resources. Examples of user tasks are writing an email, chatting with friends, watching movies over the Internet, etc. These tasks may be requested anytime, anywhere and then constructed at runtime as required by the situation in hand.

Towards this evolution, Service Oriented Architecture (SOA) has emerged as a computing paradigm that changed the traditional way of how applications are designed, implemented and consumed in a pervasive environment.

One particular approach for developing SOA-based user's tasks is to use components, which are the main building blocks in SOA. Using this approach, a user task is defined as a composition of re-usable software components, which implement the business logic of the application domain collectively by providing and requiring services to/from one another. The components required by the user task are assembled at the time of the task execution.

However in a dynamic and heterogenous execution environment, assisting users in their daily tasks by combining available networked functionalities and adapting to the specifics of pervasive environment is one of the major challenges in achieving the pervasive computing vision. Indeed, it is not certain that the user device provides the necessary components for the execution of the task. Moreover, the deployed components in the devices of the execution environments may support functionally equivalent services. This leads to look up convenient components offering the required services among devices of the execution environment. Hence, the user task can use components offered by the broad range of computing devices in pervasive environments (smartphones, PDAs, tablets, laptops, etc.) and not be limited to the services of the user device.

Building upon the SOA, it is possible to describe a user task as an assembly of abstract components (i.e., services), which are reusable software entities with well defined interfaces, and may be accessed without any knowledge about their implementations or programming languages. Thus, a user task can be crafted using a set of services. This allows the separation of the business functionality (i.e., services) from its implementation and to execute it by composing different concrete components provided by various devices.

The implementation of these services can be found by looking up concrete and deployed components in devices of the environment. Therefore, to achieve the task execution, the services of the user task should be resolved into concrete components, which are software components with specified implementations. The resolution of a user task involves an automatic selection of concrete components across various devices in the environment. A service is matched with a component if their interfaces match. A user task is said to be resolved if for all of its abstract components, we find matching concrete components implementing these services.

The complexities involved in designing and realizing such tasks have been identified and addressed by many approaches (Sousa and Garlan, 2002) (Ben Mokhtar et al., 2007) (Mukhtar et al., 2011), etc.

While existing approaches may assume that a mapping from abstract to concrete task can be done effortlessly, many problems may be captured at init time that prevent it to be achieved successfully. Moreover, components of the user task may depend on the changes of properties of components provided by the underlying environment. This arises the need to monitor their properties in order be aware about their changes. Furthermore, tasks in pervasive environments are challenged by the dynamism of their execution environment due to, e.g., user and device mobility, which make them subjects to unforeseen failures. These problems give rise to a relevant challenge, which is ensuring the continuity of execution of user tasks in such dynamic and heterogeneous environments.

1.2 Problem Description

The challenge that we are interested in this thesis, arises from the limitations of SOA to fulfil a continuous execution of applications in dynamic and heterogenous environments. To better describe the issue, let's consider a *video player* task that provides the functionality of displaying video to the user.

The task can be composed of three services (i.e. abstract components) offering functionalities namely, controlling, decoding and displaying video as shown in Figure 1.1. The *Controller* component sends a command to the *VideoDecoder* component to decode

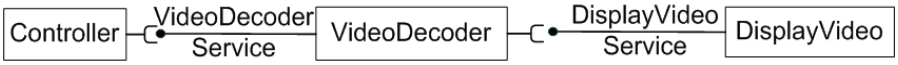


Figure 1.1: Video player task

a stored video into an appropriate format. Once the video is decoded, it is passed to the *DisplayVideo* component to play it. This is done using the service provided by the *DisplayVideo* component. The resolution of these abstract components into respective concrete ones is required for the execution of the task. It involves automatic selection of components among devices, matching with task's services.

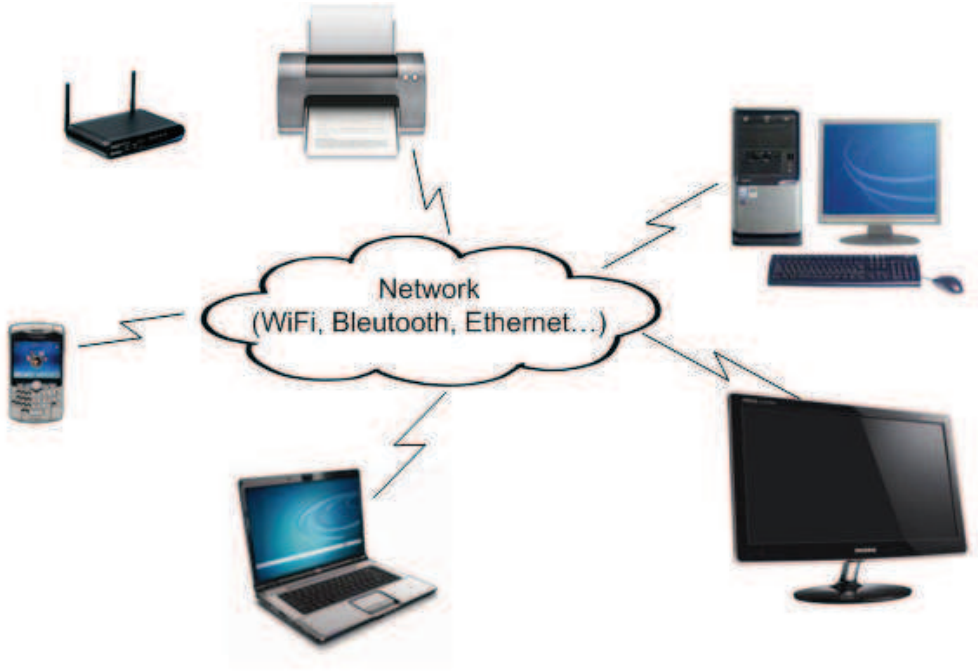


Figure 1.2: Different devices in a pervasive environment

We assume that a number of devices exist in the pervasive environment with different characteristics, ranging from small hand-held devices with limited capabilities, to powerful multimedia computers with abundant resources as shown in Figure 1.2. They are connected to each other on-the-fly using wireless communication technologies like Bluetooth, IEEE 802.11, etc. These devices may host one or more concrete components, which correspond to the services of the video player task.

For each service of the task, we may find one or more matching components across different devices. However, while these components offer similar functional interfaces, they may differ from each other in terms of the capabilities of the devices. Hence, a DisplayVideo component on a smart phone is not considered the same as one on a flat screen. Therefore, there is a need to a mechanism for a dynamic selection of a particular device among many alternatives, when they host functionally equivalent components matching with a task's services.

Moreover, it is possible that the selected device provides several components that implement the same service. For example, a laptop device provides a VLC Media Player and Real Player components, which are two media players implementing the DisplayVideo service. Thus, there is a need to carry out dynamically the selection of convenient components for the task resolution.

On other hand, the VideoDecoder component may depend on network throughput to decide the rate of frames transferred to the DisplayVideo component. In case of high throughput, video frames can be sent at higher rates. However, in case of weak throughput, smaller rate may be applied for a quick transfer. The VideoDecoder component may also depend on the changes of batteries power of the communicating devices. If the battery power of the device is low, it uses lower degree of decoding to conserve the battery power. However, if the remaining battery power of each device is above a certain threshold (e.g., 20 percent), higher rate of decoding can be used for a better quality of frames sent to the DisplayVideo component. Hence, there is a need to monitor the network throughput to decide whether the rate to apply for the video frames sent to the DisplayVideo component and the remaining batteries power to adjust the decoding rate of the VideoDecoder component.

While existing approaches may assume that a mapping from abstract to concrete task can be done effortlessly, many problems may arise at init time that prevent it to be achieved successfully. These problems imply that the user task can not be executed in the given context due to, e.g., heterogeneity of network connection interfaces, heterogeneity of interaction protocols, etc. Thus, there is a mismatching between the given abstract description of the task and the concrete level, which triggers the adaptation of the user task in order to fulfil its mapping at init time.

Assume, for example, that the VideoDecoder and DisplayVideo components are provided by two devices supporting different network connection interfaces, e.g., Bluetooth and WiFi. This means that the task can not be well executed because of the heterogeneity of network characteristics of devices. Hence, there is an indispensable need to adapt the user task at init time in order to fulfil its resolution by considering the available hardware, network and software resources in the execution environment.

Adaptation is very important also in dynamic and ever changing environments to ensure the continuity of execution of user tasks. Indeed, user tasks in pervasive environments are challenged by the dynamicity of their execution environments due to, e.g., users and devices mobility, which may make them subject to unforeseen failures. For example, if the DisplayVideo component is no longer available during the execution of the video player task, this arises the need to replace the disappeared component by another one offering the displaying functionality.

In another case, the adaptation may be triggered at runtime due to a captured mismatching between abstract description of the user task and the concrete level. Assume for example that the bandwidth of the connection interface supported by the devices selected for the execution of the video player task, becomes very weak, almost nonexistent. This change may imply a mismatch, if there is no alternative to replace the broken connection. Thus, there is a need to adapt the user task to such captured mismatches at runtime for the continuity of its execution.

We deduce a few important points from the video player task. First, the resolution of a user task to a concrete level, requires the selection of the convenient device and component for each service of the task as a pervasive environment provides a broad range of devices and components.

Second, the behavior of a user task may be dependent on certain properties of components representing the underlying environment or the user task. Thus, a monitoring mechanism is required to consider the changes of these properties irrespective of the type of components or the properties required to be monitored locally or remotely.

Finally, there is a need to a dynamic adaptation mechanism to overcome the mismatching between abstract description of the user task and the concrete level at init time or during its execution. A dynamic adaptation is required also to overcome the failures captured during the task's execution because of the devices mobility, changes of users preferences and so forth.

In summary, the need for a continuity of execution of user task give a rise to the following challenges, which are addressed in this thesis:

- **Resolution of a user task by selecting the convenient devices and components required for its execution**
- **Monitoring the changes of the execution environment**
- **Adaptation of user tasks to fulfil their execution**

1.3 Thesis Contribution and Document Structure

To address the above challenges, this thesis presents an architectural design of a middleware for tasks resolution, monitoring and tasks adaptation. The most significant contributions are structured along this document as follows:

- Chapter 2 provides an overview of work related to the continuity of executions of user tasks in pervasive environments and brings out the limitations of existing solutions. We distinguish approaches that consider abstract descriptions for applications to achieve the continuity of their execution in pervasive environments, from those that manipulate concrete descriptions for applications.
- Chapter 3, first, discuss the existing service composition approaches. Then, it details our approach for the resolution of an abstract user task that is based on the selection of a best device and component for each service of the user task (Ben Lahmar et al., 2011b) (Ben Lahmar et al., 2012). This is done by evaluating devices and components by considering some non functional constraints like user preferences, devices capabilities, services requirements and components preferences, in order to select device and component having the highest values.
- Chapter 4 deals with the monitoring of changes of an execution environment. After investigating the existing work, we propose a monitoring mechanism that allows components to be aware about the changes of the properties of other components, which they depend (Ben Lahmar et al., 2010b). If a component is not monitorable, we propose transformation mechanisms that will be applied for local as well as remote components to render them monitorable (Belaïd et al., 2010).
- Chapter 5 tackles two adaptation techniques. First technique considers the adaptation that is based on the partial reselection of components and devices with respect to the functional behaviour of the user task (Ben Lahmar et al., 2011b). The princi-

ple of this adaptation is to replace a component or device by another one whenever there is a need. The second technique of adaptation addresses the failures captured during the resolution or during the execution of a user task and that denote a mismatching between an abstract description of a user task and the concrete level. The adaptation is based on the transformation of the abstract description of the task by injecting adaptation patterns that encapsulate an extra-functional behaviour with respect to the functionalities of the task (Ben Lahmar et al., 2010a). In this chapter, we propose also a set of adaptation patterns to overcome not only software mismatches but also hardware and network mismatches (Ben Lahmar et al., 2011a).

- Chapter 6 presents an architectural design of our middleware for task resolution, monitoring of environment's changes and task adaptation. We provide details about the implementation of the middleware components along with evaluation results.
- Chapter 7 summarizes our contributions presented in this thesis and discusses further research perspectives to be explored beyond them.

Chapter 2

Continuity of Applications Execution: State of the Art

2.1 Introduction

Pervasive environment (called also ubiquitous environment and ambient intelligence), as envisioned by Weiser (Weiser, 1991), is a world where computing systems exist everywhere and allow access any time to data and computing resources. It is characterized by the interaction of a multitude of highly heterogeneous devices, ranging from powerful general-purpose servers located in the infrastructure, to tiny mobile sensors, integrated in everyday objects. It envisions the seamless applications that are cooperatively executed by integrating transparently functionalities provided by heterogeneous software and hardware resources in order to assist users in the realization of their daily tasks.

User tasks in such environments are under highly dynamic and unpredictable operating conditions. User needs may change dynamically, availability of resources may vary, devices may come and go at runtime, etc. To cope with such dynamicity and diversity,

they should have the capacity to be deployed and executed in ad hoc manner, integrating the available hardware and software resources.

Towards these objectives, software applications have evolved from centralized and stable applications, to highly decentralized and dynamic ones. This evolution has induced a change in the development of applications as described in the following.

2.1.1 Service-Oriented Pervasive Computing

Service-oriented architecture (SOA) is the outcome of the Web services developments and standards in support of automated business integration (Booth et al., 2004) (Burbeck, 2000). The purpose of this architecture style is to address the requirements of loosely coupled, standards-based, and protocol-independent distributed computing. As defined by Papazoglou (Papazoglou, 2003), *SOA is a logical way of designing a software system to provide services to either end user applications or other services distributed in a network through published and discoverable interfaces.*

The main building blocks in SOA are services. Services are self-describing, open components that support rapid, low-cost development and deployment of distributed applications. Thus, using SOA, applications are defined as a composition of re-usable software services, which implement the business logic of the application domain.

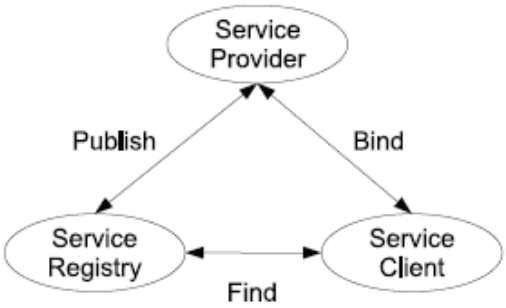


Figure 2.1: SOA architecture

The SOA architectural style is structured around the three basic actors depicted in Figure 2.1: Service Provider, Service Client and Service Registry while the interactions between them involve the publish, find and bind operations. Service Provider is the role assumed by a software entity offering a service. Service Client is the role of a requestor entity seeking to consume a specific service. However, Service Registry is the role of an entity maintaining information on available services and the way to access them.

The benefit of this approach lies in the looser coupling of the services making up an application. Services are provided by components and are platform independent, implying that a client using any computational platform, operating system and any programming language can use the service. While different Service Providers and Service Clients may use different technologies for implementing and accessing the business functionality, the representation of the functionalities on a higher level (services) is same. Therefore, it should be interesting to describe an application as a composition of abstract components (i.e., services) in order to be independent on from implementations. This allows the

separation of the business functionality (services) from its implementation (components). Hence, an application can be executed by composing different components provided by various devices with respect to their services descriptions.

However, the SOA architecture alone cannot meet the dynamicity and heterogeneity of pervasive environments. Indeed, developing and executing applications in such environments is a non-trivial task. It is possible that the Service Client depends on the changes of the Service Provider or such other services provided by the registry. This may give a rise to a monitoring need in order to be aware about their changes. Another challenge is to provide techniques and support for dynamic service compositions in that a way they equip themselves with adaptive service capabilities so that they can continually modify themselves to respond to environmental demands and changes.

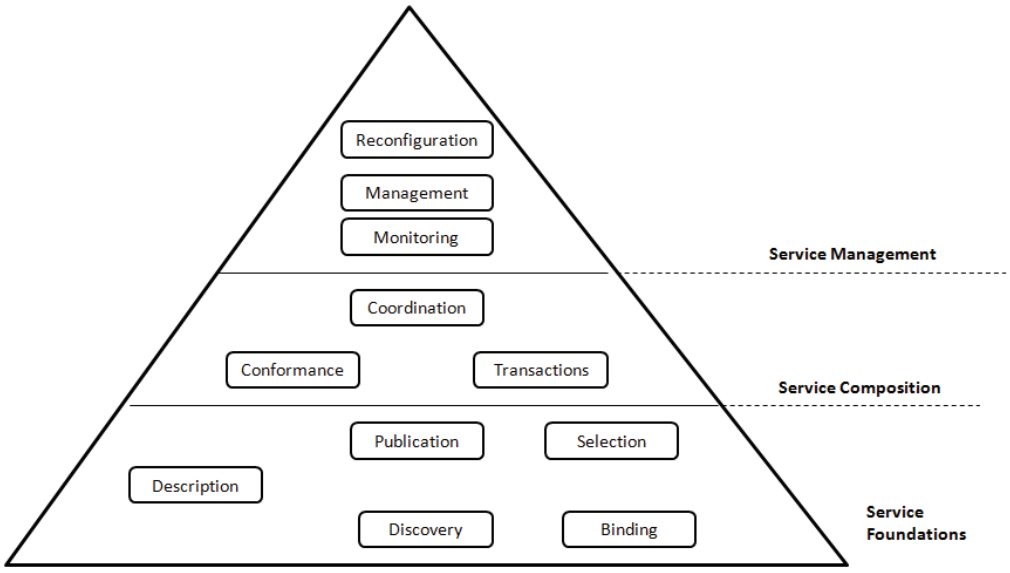


Figure 2.2: Extended SOA functionalities

In (Papazoglou and van den Heuvel, 2007), authors proposed an extension for SOA architectural model to take into account several extra functional requirements for an SOA application. They separate them into three layers, as shown in Figure 2.2, service foundations at the bottom, service composition in the middle, and service management and monitoring on top:

- Service Foundations consists of a service-oriented middleware backbone that realizes the runtime SOA infrastructure. This infrastructure connects heterogeneous components over various networks by publishing, finding, and binding of services as described in Figure 2.1.
- Service Composition is responsible for combining multiple services into a single composite service at design-time (static) or at run-time (dynamic). Resulting composite services can be used as basic services in further service compositions or offered as complete applications and solutions to Service Clients.
- Service Management requires to realize monitoring and management activities over the services. This includes the collection of information about the SOA-based application during its life cycle, and to observe the execution status of composite

services to possibly trigger service adaptation. Management activities also include the modification of the composition of the services in order to adapt to some specified conditions.

2.1.2 Autonomic Computing

Autonomic computing aims to manage the computing systems with decreasing human intervention. The term autonomic is inspired from the human body where the autonomic nervous system takes care of unconscious reflexes, the digestive functions of the stomach, the rate and depth of respiration and so forth (Huebscher and McCann, 2008). Autonomic computing attempts to intervene in computing systems in a similar fashion as its biological counterpart. The term autonomic was introduced by IBM in 2001 (Horn, 2001). This initiative aims to develop computer systems capable of self-management, to overcome the rapidly growing complexity of computing systems management, and to reduce the barrier that complexity poses to further growth.

Management tasks like monitoring, configuration, protection, optimization, are not the main functional objective of most applications, but if they are not properly addressed, the application cannot accomplish its task. The challenge, then, is to enable self-managing systems that take control of all these non functional tasks, letting the developers to focus on the main functional goals of applications. In order to really free developers from the burden of programming self-governing features on their applications, there must exist a way to develop these concerns independently and to integrate them with the application at some stage.

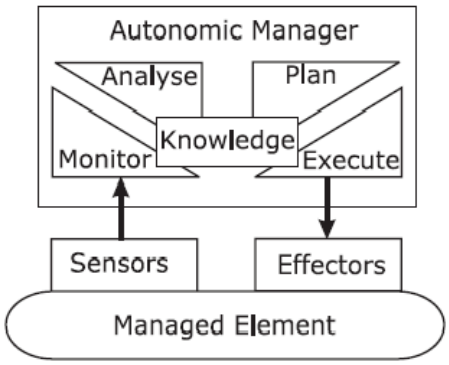


Figure 2.3: Autonomic control loop (Computing et al., 2006)

To achieve autonomic computing, IBM has suggested a reference model for autonomic control loops (Computing et al., 2006) as depicted in Figure 2.3. It is called the MAPE-K (Monitor, Analyse, Plan, Execute, Knowledge) loop. The central element of MAPE-K is the autonomic manager that implements the autonomic behaviour. In the MAPE-K autonomic loop, the managed element represents any software or hardware resource that is coupled with an autonomic manager to exhibit an autonomic behaviour. Sensors are used to collect information about the managed element, while effectors carry out changes to the managed element. Sometimes also a common element called Knowledge source is highlighted, which represents the management data that can be shared for all the phases

of the control loop. Knowledge include for example, requests for change, change plans, etc. Thus, based on the data collected by the sensors and the internal knowledge, the autonomic manager will monitor the managed element and execute changes on it through effectors.

The phases of the autonomic control loop are defined as: Monitor, Analyse, Plan, and Execute.

- The Monitor function that provides the mechanisms to collect, aggregate, filter and report monitoring data collected from a managed resource through sensors.
- The Analyze function that provides the mechanisms that correlate and model complex situations and allow the autonomic manager to interpret the environment, predict future situations, and interpret the current state of the system.
- The Plan function that provides the mechanisms that construct the actions needed to achieve a certain goal, usually according to some guiding policy or strategy.
- The Execute function that provides the mechanisms to control the execution of the plan over the managed resources by means of effectors.

The use of autonomic capabilities in conjunction with SOA provides an evolutionary approach in which autonomic computing capabilities anticipate runtime application requirements and resolve problems with minimal human intervention.

The challenge is to ensure that applications can operate over devices using multiple wireless access networks by selecting the most appropriate services for its execution, to be aware about the changes of execution environments and to adapt the applications once changes are captured. These give a rise to a crucial issue, which is ensuring the continuity of execution of applications in such dynamic and heterogenous environments. This challenge is divided into three subproblems namely, services composition, monitoring of pervasive environments and services adaptation.

Services composition aims to realize a user task by combining multiple services into a single composite service at design-time (static) or at run-time (dynamic). The challenge is how to realize that task regarding the diversity of devices and services in pervasive environments.

The monitoring is an essential issue when dealing with such dynamic and heterogeneous environments in order to be aware about their changes that may have an impact in the realizations and executions of user tasks.

Once changes are captured, there is a need to trigger an adaptation for the continuity of execution of applications. In the literature, we distinguish two relevant adaptive techniques used in pervasive environment (McKinley et al., 2004). Parametric adaptations aim at adjusting internal or global parameters in order to respond to changes of the environment. Compositional adaptation is classified into structural and behavioural adaptations. A behavioural adaptation implies the modification of the functional behaviour of the application in response to changes in its execution environment. However, structural adaptation allows the restructuring of the application by adding or removing software entities with respect to its functional logic.

As it can be seen, these aspects correspond to the second and third layers of the extended architectural model of SOA as defined in (Papazoglou and van den Heuvel,

2007). Regarding the requirements for the continuity of user tasks executions, the autonomic control loop cover the different aspects that we are looking for. The monitoring and analysis phases are needed to notify components about the changes of the execution environment that seem to be interesting for their execution. The execution phase correspond either to the composition or to the adaptation actions. The planning phase is required to determine which actions will be performed to accomplish the services composition or adaptation. However, in this thesis, we do not carry about the planning phase that remains as challenge in the future work.

In this chapter, we investigate how each of the existing work deals with the problem of the continuity of execution of applications in pervasive environments. The chapter is structured as following. In Section 2.2, we present approaches that consider abstract descriptions of applications for dynamic composition, monitoring and adaptation like Aura (Sousa and Garlan, 2002), Gaia (Román and Campbell, 2003), etc. Then, we introduce in Section 2.3 approaches manipulating concrete descriptions of applications to achieve the continuity of their executions like MADAM (Floch et al., 2006), ReMMoc (Grace et al., 2003), etc. In Section 2.4, we provide an overall discussion and classification of the cited work.

2.2 Approaches for Abstract Applications

Approaches in this category deal with the continuity of execution of abstract applications. An abstract application is represented by an assembly of abstract components that specify only their offers and requirements. In this section, we present the different aspects of these approaches to ensure the composition, monitoring and adaptation challenges.

2.2.1 Aura

The Aura project (Sousa and Garlan, 2002) defines an architecture for home and office environments to allow users to dynamically realize their tasks that are represented as a set of abstract services.

Aura is based on the concept of a personal Aura. A personal Aura acts as a proxy for the mobile user. When a user enters a new environment, the Aura marshals the appropriate resources in the environment to support its task. Furthermore, an Aura captures the physical context around the user like user location, other people in the vicinity, etc.

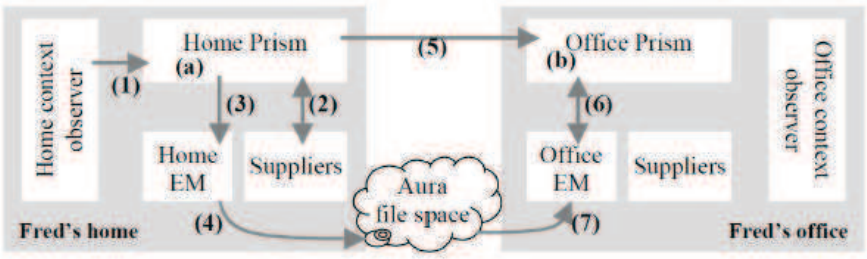


Figure 2.4: Aura architecture (Sousa and Garlan, 2002)

The Aura project proposes an architecture for configuration and reconfiguration consisting of four architectural components as shown in Figure 2.4. First, the Task Manager, called Prism, is used to embody the concept of personal Aura. It captures knowledge about user's tasks and associated context. Based on this information, the Task Manager is responsible for the configuration and reconfiguration of the environment to best serve the user. Second, the Context Observer provides information on the physical context and reports relevant events back to Prism. Third, the Environment Manager embodies the gateway between the environment-independent requests made by the Task Manager and the concrete applications and devices (Suppliers) of the environment layer. Suppliers provide abstraction of applications and devices; they are employed by the Environment Manager to support a user's task. When Suppliers are installed in an environment, they become registered with the local Environment Manager. Such a registry is the base for matching requests for services.

The mapping of Suppliers consists of wrapping of existing applications and services to conform to Aura APIs. Such wrappers play a fundamental role while instantiating a task by considering the user preferences and the available resources (Sousa et al., 2006). User preferences are expressed as multidimensional utility functions having three parts. First, configuration preferences capture preferences with respect to the set of services to support a task to reflect how happy the user is with each possible set of services interconnections. Second, supplier preferences capture which specific suppliers are preferred to provide the required services and how happy the user is with the choice of a specific supplier for a specific service. And third, QoS preferences capture the acceptable QoS levels and preferred services. Based on the the supplier and QoS preferences, the wrappers map the abstract service descriptions into application-specific settings to maximize the utility function.

The Environment Manager adjusts such mapping automatically, not only in response to changes in the user's needs, but also in response to changes in the environment's capabilities (adaptation initiated by the Environment Manager).

Indeed, Aura deals with three contexts triggering the adaptation of a user task. First, as users move from one location to another, there is a need to migrate the information. This is ensured by the Task Manager that coordinates the migration of all the information related to the user task to the new environment. Second, the Task Manager coordinates the suspension of the executing task, if users switch from one task to another, or resume previous tasks by finding and configuring suitable services to support their tasks. Third, the infrastructure shields the user as much as possible from distractions by automatically adapting to dynamically changing resources.

The adaptation is achieved by the Environment Manager by reselecting new services implementations given the existing resources by considering the user preferences towards the reconfigurations. In this case, the reconfigurations are ranked according to the utility function in order to select the reconfiguration that maximizes the utility.

For example, Figure 2.4 illustrates an example of a user switching between two locations, where the Aura environment at Fred's home cooperates with the Aura environment at his office to migrate tasks between the two locations. When Fred leaves one environment, the local Context Observer points out that event to the Task Manager. The Task Manager then checkpoints the state of the running services in a platform-independent fashion and causes the local Environment Manager to pause those services. This informa-

tion, along with Fred's task state, is stored in a distributed file space. When Fred enters his office environment, the local Context Observer notices the event and informs the local Task Manager. The Task Manager reinstantiates the tasks by finding and configuring service Suppliers in the new environment. This reconfiguration is achieved by calculating and selecting the best combination of suppliers that can provide the required services, satisfy the user's configuration preference, and maximize the utility function.

2.2.2 Gaia

Gaia (Román et al., 2002) is a distributed middleware infrastructure that enables the dynamic execution of software applications in a physical space, called an active space. An active space represents an integrated habitat that merges physical and computational infrastructures therein.

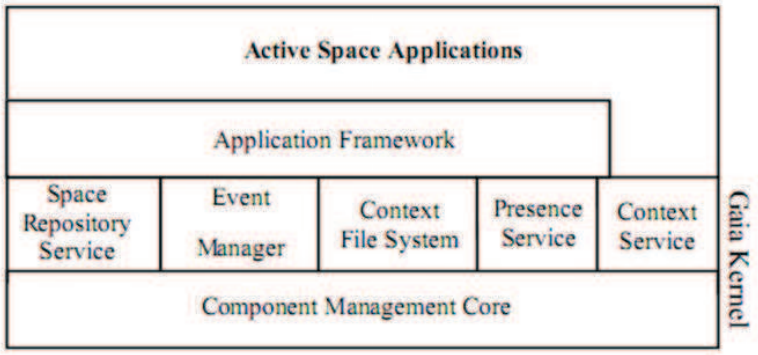


Figure 2.5: Gaia architecture (Román et al., 2002)

Figure 2.5 shows the three major building blocks of Gaia: the Gaia Kernel, the Gaia Application Framework, and the Applications.

The Gaia Kernel contains a Component Management Core and a set of basic services that are used by all the Gaia applications. The Component Management Core is responsible for load, unload, transfer, creation, and destruction of applications' components.

Gaia contains five basic services which are the Event Manager Service, Presence Service, Context Service, Space Repository Service, and Context File System. The Event Manager Service distributes events in the active space and implements a decoupled communication model based on suppliers, consumers, and channels. The Context Service provides information about the current context. It also allows applications to query and register for particular context information so that they may adapt to their environment. The Presence Service detects digital and physical entities present in an active space like application, service, device, and person. The Space Repository Service stores information about all software and hardware entities contained in the space (e.g., name, type, and owner) and provides functionality to browse and retrieve entities based on specific attributes. The Context File System incorporates context into the traditional file system model to provide support for mobile users, device heterogeneity, and data organization.

Gaia's applications use a set of component building blocks, organized as Application Frameworks (Román and Campbell, 2003), to support applications that execute within an active space. The Application Framework defines two types of application descriptions:

the application generic description (AGD), and the application customized description (ACD). The AGD is an active space-independent application description that lists the components of an application and their requirements. This description is mapped later to an ACD that uses resources present in the active space. The ACD matches the application requirements listed in the generic description. The mapping mechanism uses the requirements to query Gaia OS to obtain a list of matching entities. It can be a user-assisted or automatic.

Users can describe tasks to be performed in terms of abstract goals and an application framework decides how these goals can be achieved (Ranganathan and Campbell, 2004). The key element of the architecture of the framework is the Planning Component. The user indicates his goal to the Planning Component through a GUI. The framework then analyzes the different choices available for achieving user goals based on the user's preferences and the current context. It then plans a sequence of actions to achieve the goals and executes them.

Gaia uses a STRIPS planning algorithm (Ranganathan and Campbell, 2004) to implement the Planning Component. The planning algorithm takes in an abstract goal specification, generates a template goal state and then converts that into a concrete goal state by considering into account the context and the user preferences. Each state of the environment is associated with a utility function. This allows the planning framework to compare different goal states and choose the best one with the maximum utility.

The planning framework supports also the dynamic reconfiguration of applications. It deals with actions failure that is detected if the entity on which the method is invoked is not reachable or does not respond. If a failure is detected, the planning framework invokes the same action once again, else it replans to get to the next best goal state. Gaia also allows changing the composition of an application dynamically upon a user request. For example, a user may specify a new device providing a component that should replace a component currently used. Furthermore, Gaia supports two different types of mobility: intra-space mobility and inter-space mobility. Intra-space mobility is related to the migration of application components inside an active space and is the result of application partitioning among different devices. Intra-space mobility allows users and external services to move application components among different devices. Inter-space mobility concerns moving applications across different spaces.

In Gaia, event-based communication is used for notifying about changes in the active space or in the state of components, for both kernel and application components.

2.2.3 SeSCo

SeSCo (Seamless Service Composition) provides a mechanism for dynamic service composition in pervasive environments (Kalasapur et al., 2007). This mechanism implements a process for resolving an abstract specification of a task to a concrete services composition.

The proposed service composition mechanism employs the service-oriented middleware platform called Pervasive Information Communities Organization (PICO) (Kumar et al., 2003) to model and represent resources as services. Using PICO, services are modeled as directed attributed graphs. Each service is described as a simple graph where a directed edge represents the inputs and outputs for the service. The edge attributes

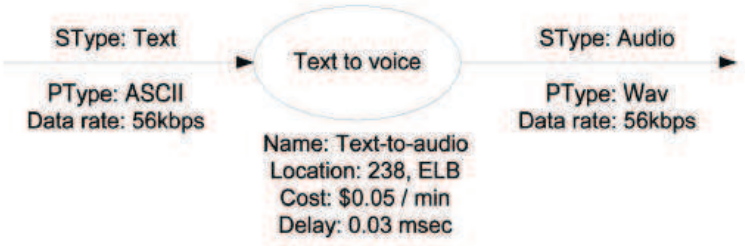


Figure 2.6: Representation of a service (Kalasapur et al., 2007)

include the semantic description of the parameter, the parameter type, the data rates, formats, and so forth. Figure 2.6 shows the representation of a service that performs text-to-voice conversion. The semantic attribute associated with the input is text, whereas the syntactic attribute specifies the expected form of text, which is ASCII in this case, along with other associated parameters such as the data rate.

A task is submitted for resolution to the directory that can have a centralized or a distributed structure. In the directory, the components of the task graph are matched against available basic services to generate possible composite services. Indeed, the service graphs are parsed for storage into a two-layered aggregation graph. For each registered service, the first stage of aggregation is based on the semantic parameters associated with the service. The second stage of aggregation is based on the syntactic type of the parameters.

The task resolution is also performed in two steps. In the first step, one or more possible compositions are derived at the semantic level based on the aggregated graph. If a composition is possible at the semantic level, the underlying services that can take part in the composed result are identified at the second level of aggregation during the syntactic matching. The shortest path will be selected for the mapping of the task. This is done by considering requirements such as the permissible delay, cost of utilization, location, and so forth. These requirements are specified within the task through the attributes on the nodes and treated as weights along the edges. The computation of the shortest path has to consider these multiple constraints along the edges as weights.

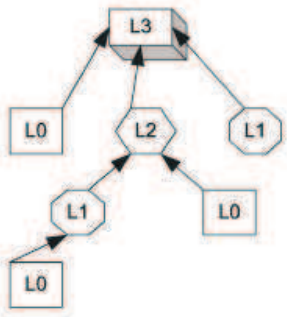


Figure 2.7: Hierarchical organization of devices (Kalasapur et al., 2007)

To achieve the mapping upon a distributed directory, they propose a classification of devices in four levels as shown in Figure 2.7. Resources with relatively higher degrees of availability can be classified into the highest end of the spectrum, which is a level 3 device like PC, servers, etc. User devices like laptop or PDA are classified under level 2. Level 1 include mobile resources like cell phone and so on. Level-0 devices are resources with no native support for additional configuration, e.g., sensors, printers, etc. The classification of resources into one of the above levels is performed at the time of configuring the resources.

Services composition is based on a device overlay formed through a latch protocol. The basic principle of latching is that a device with lower resource availability latches itself to another device with higher resource availability. Thus, devices are structured as a tree that implies the device levels with higher devices up in the tree and lower devices latched to them. The result of the latch process is a service zone where a device includes all its children of within the hierarchy and itself. The search zone of the device is its own service zone if a device level is higher than 1. Otherwise, the search zone of device is the service zone of its parent. When a device has a task to be composed, the aggregation in the local search zone is first inspected. If any of the required services for the task cannot be met, the search zone is then expanded to the search zone of its parent, and so on.

Due to the dynamic nature of pervasive environments, the structure of the hierarchy can change frequently. The change in the structure can be triggered by disappearance of a device or the appearance of a new one. When a new device joins the hierarchy, it sends a $LATCH\_HELO$ message to the neighboring devices to advertise its appearance, along with the information about its level. It will be added as either a parent, child, or sibling based on its level. When a device leaves a parent, it may notify the parent of its departure. Otherwise, the parent decides that the device is no longer available due to the missing periodic $LATCH\_HELO$ messages. Thus, it is necessary to recompute a portion of the composition to fulfill the part that was being played by the missing service. Based on the hierarchy, a new composition can be generated to meet the requirements of the affected node by looking up the search zone.

2.3 Approaches for Concrete Applications

In this section, we present some research work dealing with the continuity of execution of applications that are described as an assembly of concrete components. We mean by a concrete component, any component whose description specifies its implementation. We illustrate how these work tackle the different aspects related to the continuity of execution; which are namely composition, monitoring and adaptation.

2.3.1 ReMMoC

In mobile environments, services can be developed upon a range of middleware types (e.g. RMI and publish-subscribe) and advertised using different service discovery protocols (e.g. UPnP and SLP) unknown to the application developer. Towards this challenge, the ReMMoC middleware (Reflective Middleware for Mobile Computing) (Grace et al., 2003), which is an adaptation middleware, enables software applications to be developed independently of specific middleware technologies. Thus, applications are able to dis-

cover and interoperate with a range of heterogeneous services available in the execution environment.

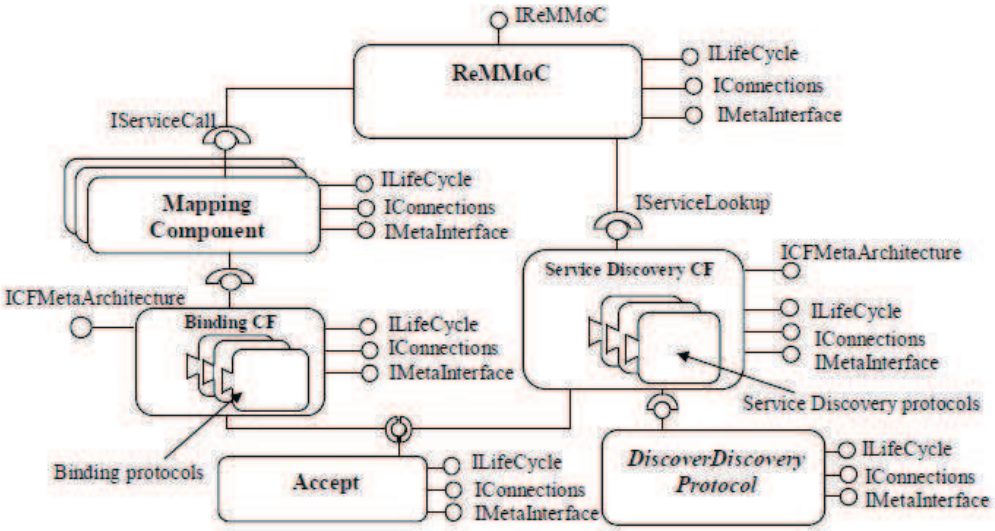


Figure 2.8: ReMMoC middleware (Grace et al., 2003)

The ReMMoC middleware is built on OpenCom component model (Coulson et al., 2004), as shown in Figure 2.8. This component model is used to construct families of middleware that are constructed as a set of configurable component frameworks. It uses reflection to discover the current structure and behaviour of framework, and to enable selected changes at run-time.

Built upon OpenCOM component model, ReMMoC middleware consists of two OpenCom frameworks, a binding framework for interoperation with mobile services implemented upon different middleware types and a service discovery framework for discovering services advertised by a range of service discovery protocols. A component framework in OpenCOM is itself an OpenCOM component that maintains internal structure to implement its service functionality. An OpenCOM Component framework implements the base interfaces of an OpenCOM component (IMetaInterface, ILifeCycle, IConnections) in addition to its own services interfaces and receptacles (i.e., required services). These interfaces allow the inspection of the current structure of the framework and the ability to dynamically alter its behaviour. ILifeCycle offers operations to be called by the run-time when an instance of the host component is created or destroyed. IMetaInterface supports inspection of the types of all interfaces and receptacles declared by the component framework. However, IConnections offers methods to modify the interfaces connected to a framework's receptacles. Each component framework is added by the ICFMetaArchitecture interface, which provides reflective operations to inspect and dynamically reconfigure the framework's local component architecture.

Each component framework of ReMMoC middleware is able to: 1) find the required mobile services irrespective of the service discovery protocol and 2) interoperate with services implemented upon different interaction types. The framework monitors the envi-

ronment and the service types in use and reconfigures itself to mirror the current setup. The reconfiguration consists of components replacement or the framework restructuring by adding or removing components. Hence, the binding framework allows services to interact by plugging in different binding type implementations, such as IIOP (Internet Inter-Orb Protocol), publish / subscribe, and SOAP. However, the service discovery framework allows the plugging in service discovery protocols (for example, SLP and UPnP) that are being used to advertise services.

The adaptation is ensured by using the MOPs (Meta Object Protocols) of an OpenCom component that is based on reflection. MOPs provide a set of methods to introspect and adapt this meta-representation by adding or removing protocols or a binding type from a framework at runtime.

The ReMMoC component, as shown in Figure 2.8, performs reconfiguration management and provides a generic API that are mapped to the technology specific APIs. For example, when a service is advertised using the abstraction API, this is mapped onto the underlying protocols e.g., SLP or UPnP. But if SLP and UPnP are configured then the service will be advertised using both.

2.3.2 MADAM

The middleware MADAM (Mobility and ADaption enAbling Middleware) (Floch et al., 2006) aims to facilitate adaptive application development for mobile computing. In MADAM, a component is defined as a unit of composition with contractually specified interfaces and explicit dependencies. A component type is associated to a set of port types that components should implement. A port represents the component's capability of participating in a specific interaction.

To support adaptation, a runtime representation architecture model is required to allow middleware components to reason about and control adaptation. The architecture model includes specification of the application structure, the application's variability and the properties of each variant.

MADAM architecture model uses component frameworks to design applications. In MADAM, a component framework describes a composition of component types that are achieved by plugging in different component implementations conforming to the type. To discriminate between alternative component implementations in MADAM architecture models, components types are annotated with properties in order to compare their implementations. Properties qualify the services that components offer or need. Because sometimes the properties cannot be expressed by constant values, they propose to use property predictors, which enable the expression of a property as a function of other properties and possibly context values.

In addition to the framework architecture model, the middleware also builds and maintains an instance architecture model, which is a model of the running application variant. The context model is used to represent the context of execution of applications.

Figure 2.9 shows the MADAM middleware architecture, which consists of four components: Context manager, Adaptation manager, Configurator and a Core.

A Context manager component is responsible for context reasoning, such as aggregation, derivation in order to provide the Adaptation manager component with relevant context information when context changes occur. The Adaptation manager is responsi-

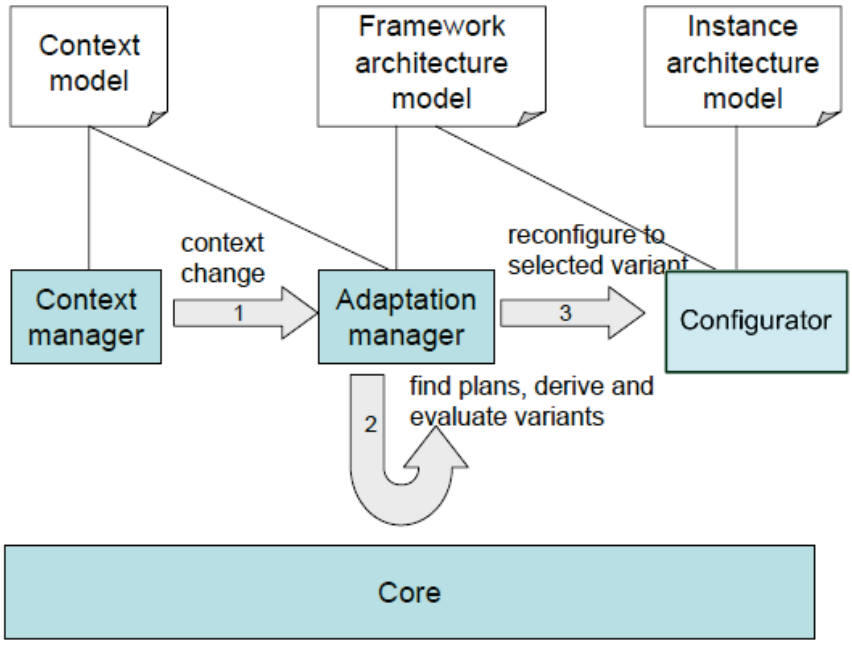


Figure 2.9: Madam middleware architecture (Floch et al., 2006)

ble for reasoning on the impact of context changes on the application, and for planning. The Configurator component is responsible for reconfiguring an application by deleting or replacing component instances, instantiating components, transferring states, etc. The middleware Core component provides platform-independent services for managing applications, components, and component instances. This includes operations for publication and discovery of component frameworks and implementations and for loading, unloading, and connecting components.

Indeed, MADAM deals with two types of variation for applications, namely compositional variability and parameter variability. Compositional variability designates places in a component framework architecture where one or multiple, logically alternative elements can be plugged in and composed dynamically. Parameter variability includes tuning mechanisms to modify programs variables and behaviour which can be suitable for fine-grained adaptation such as buffer size, device parameters, etc.

The decision of which adaptations to make for variability, is done by the Adaptation manager by planning the reconfigurations. The planning consists of dynamic discovery of implementation alternatives at the variation points of the application's component framework, and further of selection of those that best matches the operational environment and provides the highest user satisfaction given the current user context. A plan is specified by the application developer. It may refer to a composition of component types as well as to an implementation or to a property predictor function.

The Adaptation manager evaluates the variants by calculating the utility of each variant in the current context. An utility function is specified by the developer regarding components properties and user requirements. If the adaptation manager determines that a variant has a better utility than the current instance, a reconfiguration takes place.

2.3.3 PCOM

PCOM (Pervasive Component system) (Becker et al., 2004a) is a component system that allows specification of a distributed application and supports automatic application adaptation. The PCOM architecture, shown in Figure 2.10, is structured in two well differentiated parts. On one hand, the lower part, named BASE, manages the communication among devices. On the other hand, the upper part, named PCOM, offers high-level programming abstraction to application programmers.

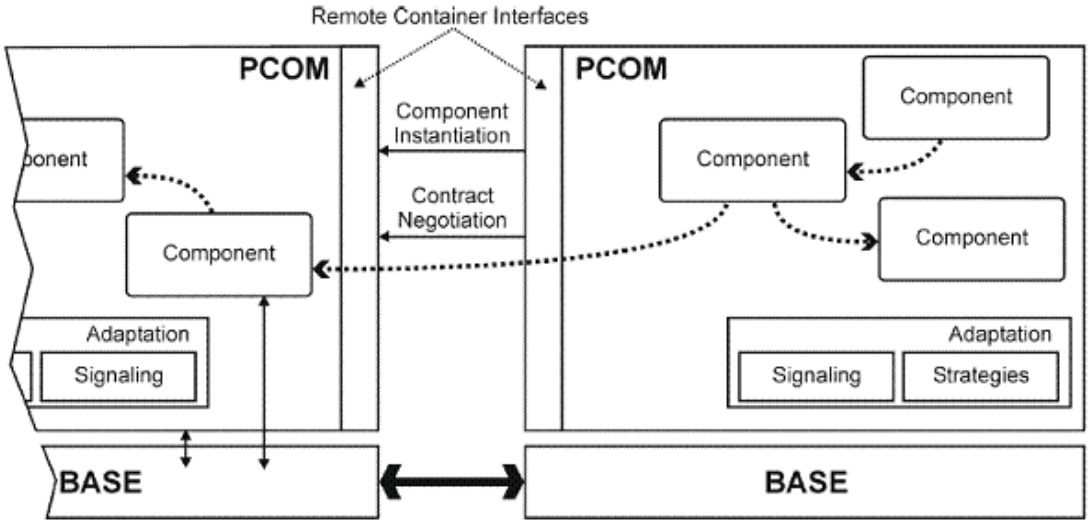


Figure 2.10: PCOM architecture (Becker et al., 2004a)

PCOM applications are defined in terms of components, which dependencies are explicitly specified as contracts. A contract defines the characteristics of the component in terms of its implementation, its offer, and its requirements with respect to other components. Components reside within a component container that is running on every device to manage their dependencies, and thus acts as a distributed execution environment for applications.

A PCOM application is modeled as a tree of components where the root component (called application anchor) identifies the application. The application tree reflects the dependencies between components where the successors of a component identify its dependencies in order to fulfill the service. The life cycle of an application is reflected by the life cycle of its application anchor.

A configuration algorithm that is cooperatively executed by the containers of an environment ensures that only valid configurations are started. A valid configuration is thereby defined as a configuration that does not have unresolved dependencies. Before instantiating a component, PCOM examines its contract dependencies and tries to find the required components. Dependencies on local resources are automatically resolved by the container that hosts the component. To resolve remote component requirements, each container is equipped with a remote query interface that lets another container search for matching offers.

The PCOM containers provide three signaling mechanisms to detect changes of the environment. A communication listener mechanism was handled to detect the disappear-

ance of components. To capture the arrival of new components, a discovery listener is used to discover the new components deployed in devices. The last signaling mechanism provided by PCOM container is contract listeners that are notified whenever a parameter changes.

PCOM deals with contractual and compositional changes. Contractual changes express that a previously agreed set of properties can no longer be guaranteed, e.g. due to changed network properties. Compositional changes are a result of device mobility or failures.

To react to these changes, PCOM allows a parametric adaptation by adjusting components' properties. It allows also a compositional adaptation by replacing components that are no longer available. The replacement is ensured by escalating to a higher level of the tree. The replacement of a sub-tree starts from the parent of the component and may include its predecessor if necessary. The escalation continues until a component resolves the conflict by reselecting components.

PCOM containers are equipped with an algorithm that performs resource aware application configuration and adaptation. To decrease the communication required to compute a reconfiguration, an architectural extension to PCOM is proposed in (Handte et al., 2007) in order to enable the system to switch between configuration algorithms at runtime.

The configuration algorithm is fully distributed, i.e. each component container configures and adapts its hosted components. For this goal, they have extended the container with two services: assembler and the application manager. The application manager is responsible for managing the life cycle of the application and the selection of the configuration. However, the assembler is responsible for computing a valid configuration.

Once the configuration is computed, the assembler can determine which dependencies need to be resolved in order to transform the current invalid configuration into a valid one. To compute a valid configuration, the assembler needs to be able to determine the set of resources that are available on each device and to find the components that can be used to resolve a dependency. Then, the container retrieves the configuration for each child component required by the anchor. Using the configuration, the container decides whether the child must be reused or replaced.

2.3.4 MySIM

Service Integration Middleware for pervasive Environment(MySIM) is a distributed middleware for a spontaneous services integration in pervasive environment (Ibrahim et al., 2009). The focus is to allow a spontaneously transformation, composition and adaptation of services.

The middleware, as shown in Figure 2.11, consists of four modules: the Translator, the Generator, the Evaluator, and the Builder.

The Translator module enables a service expressed and provided in a predefined technology to be transformed into a generic service model. A component defined by the generic service model is composed of : an interface, an implementation and QoS non functional properties. A functional interface specifies operations that can be performed by a set of inputs and outputs. The implementation is of the operations defined in the functional interface. The QoS non functional properties describe the operation capabilities. These capabilities reflect the quality of the functionality expected from the service,

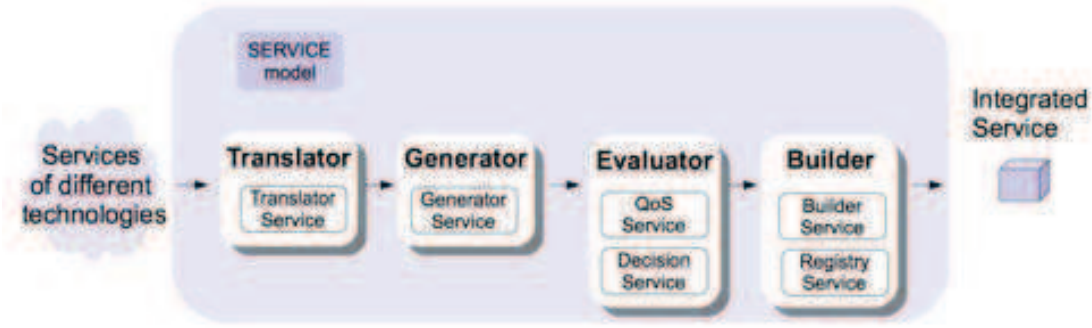


Figure 2.11: MySIM middleware (Ibrahim et al., 2009)

such as dependability (including availability, reliability, security and safety), accuracy of the operation, speed of the operation, and so on.

The Translator Service provides rules to map from OSGi services (OSGI, 1999), and Web service descriptions to the generic service model. The transformation is ensured by extracting and identifying the three main parts of a service: the interface by extracting the operation signatures and semantic description, the implementations by extracting the operation implementations and the QoS non-functional properties by extracting the non functional properties.

The transformation is essential to compose or to adapt services once they are expressed in the same model. For these objective, the service is sent to the Generator module once it is translated. The Generator Service is responsible of the syntactic and semantic matching of the functional interfaces of services in order to compose, substitute or adapt services. It tries to generate one or several composition or adaptation plans based on syntactic and semantic matching.

In the case of composition, the Generator Service finds, for each service, all the services that respond to a composition relation, syntactic or semantic, between the functional parts of services.

The adaptation of services composition is triggered either by the disappearance of a selected service or the appearance of a new one that seems to be interesting for the application execution. MySIM reacts to the appearance and disappearance of services by spontaneously integrating services into their new inhabitant in a completely transparent way for users and applications. In this case, the Generator Service finds all the services available in the environment, which are syntactically or semantically equivalent to the disappeared or the new service for a possible substitution.

Then, these plans will be evaluated by the Evaluator module in order to select the best plan to accomplish a composition or an adaptation process. The QoS service of the Evaluator module checks for every service composition, that the non-functional QoS properties of inputs and outputs of service operations are equivalent. In a service adaptation, the QoS Service assures a substitution between equivalent services by choosing the best QoS properties of services for a given application. For this goal, MySIM uses a metric that measures the non-functional QoS degree of QoS similarities between syntac-

tically and semantically matched services. The Decision Service of the Evaluator module is responsible of the plan selection.

The last step is achieved by the Builder module that is responsible for executing the real service integration and registering. The Builder Service creates new functionalities respecting the service model and directly implements these services in a chosen technology model in order to allow its integration. The Builder module provides also a Registry Service to register the interfaces of the newly transformed or/and composed services in the environment. It monitors these services by checking periodically if they execute correctly. Otherwise, it notifies the Decision Service of the Evaluator module of the events related to service appearance or disappearance.

Thus, MySIM returns new services with the same well known interfaces, but different implementations, and non-functional QoS properties in order to allow applications not only to use the available services in their vicinity but also to compose and adapt services.

2.4 Discussion

We have investigated previously how each of these state-of-the-art solutions are applied for the continuity of execution of applications in pervasive environments. The continuity of execution of applications in pervasive environments as we define it, is composed of three complementary sub-problems: composition, monitoring and adaptation. Some of the approaches focus on a specific aspect, while some others deal with more than one aspect. We compare the cited work to distinguish the ones that ensures a complete or partial continuity of execution of applications in pervasive environments as shown in Table 2.1.

	Composition	**Monitoring**	**Adaptation**
Aura	√	√	√
Gaia	√	√	√
SeSCo	√	√	√
ReMMoC			√
MADAM	√	√	√
PCOM	√	√	√
MySIM	√		√

Table 2.1: Approaches classification

Aura middleware (Sousa and Garlan, 2002) (Sousa et al., 2006) presents an architecture for resolution of abstract services of user tasks by finding suppliers that can seamlessly provide specified services. More specifically, Aura calculates and selects the best combination of suppliers that can provide the required services. The resolution of the abstract user's task is done by wrapping existing applications and services regarding the available resources and user preferences. However, wrapping a complex application does not seem to be a trivial task as it depends on the applications' complexity. The monitoring of the execution environment is ensured by the Environment Manager that detects the changes of the context. For the adaptation aspect, Aura allows the replace-

ment of services implementations when moving from one location to another by ranking the configurations given an utility function.

Gaia middleware (Román and Campbell, 2003) (Ranganathan and Campbell, 2004) deals too with the composition issue for users tasks described as a set of abstract services. To realize a Gaia application, they propose a planning algorithm that computes a concrete realization by considering the user preferences and the execution context. For the adaptation aspect, Gaia considers the failure of actions invocations as triggers for the adaptation of an application. The reconfiguration is achieved by recalculating a new combination that maximize the user preferences. In Gaia, event-based communication is used for notifying about changes in the active space or in the state of components, for both kernel and application components.

SeSCo (Kalasapur et al., 2007) proposes an architecture for realizing abstract descriptions of users tasks based on the latching protocol. The latching protocol selects devices with high levels availability for the services execution. Thus, this selection may prevent devices with low level to be selected for the execution of the task even if they respond better to services requirements. Moreover, the services composition does not consider user preferences for the task execution and it is limited to services semantics and requirements like cost of utilization and location, and so forth. For the adaptation aspect, SeSCo deals with the appearance or disappearance of devices events as triggers for the adaptation of a user task. These events are detected by sending latching messages that denote the arrival or departure of devices. However, the reconfiguration in pervasive environment may be triggered also by the changes of devices capabilities which may have an impact on devices levels and hence the selection of devices. Towards the appearance and disappearance of devices, they claim in their work that they ensure a portion recomposition of the affected part of the task by reselecting devices without further details.

The ReMMoC middleware (Grace et al., 2003) focuses on the adaptation feature and it proposes an adaptation approach that enables applications to be developed independently of specific middleware technologies and to interoperate between them. Specifically, upon the detection of the specific service discovery and access protocols employed in the current environment, ReMMoC reconfigures by loading the appropriate component frameworks enabling service requesters to use those protocols. However, the ReMMoC middleware is limited to the reconfiguration of service discovery and service interaction protocols. Moreover, they do not specify how the middleware detects the used protocols in an execution environment.

MADAM (Floch et al., 2006) presents an architecture that covers the different aspects of the continuity of execution of applications. In MADAM, components are composed and configured by using plans. The plans are composed based on the type compatibility of components to describe alternative application configurations. Then, the application configurations are ranked by evaluating their utility with regards to the application objectives. However, these configuration plans are predefined by a developer at design time, which limits the possible adaptations to these predefined plans. Moreover, the utility function is limited to the application and hence can not be used for other applications. To monitor the changes of context, MADAM provides a context manager component that it is responsible for context reasoning.

The PCOM system (Becker et al., 2004a)(Schuhmann et al., 2008) allows specification of distributed applications that is modeled as a tree of components. A PCOM-based

application is realized by resolving the contracts of its components by considering user preferences. PCOM also considers adaptation of applications by replacing a subtree if a component if it is no longer available or if a new component appears and seems to be interesting for the application execution. For the monitoring feature, each PCOM container provides signaling mechanisms to monitor only its provided components.

Using MySIM middleware (Ibrahim et al., 2009), services are translated using a generic model in order to be composed if they are syntactically and semantically matched. For this, the services composition considers the non-functional QoS (e.g., reliability, security, etc) that does not include the user preferences. Moreover, they claim in their work that the Translator component is able to transform any service independently of its description. For the adaptation aspect, MySIM considers only the appearance or disappearance of components to trigger adaptation of services. Towards these changes, it recalculates new reconfigurations to reselect components. However, in their work, they do not specify any monitoring mechanism to capture these changes.

As it can be seen, few of work consider the three aspects for the continuity of applications in pervasive environments. Moreover, most of them deal with concrete descriptions to fulfil the composition, monitoring and adaptation of applications therein. Towards the composition challenges, most of the cited work considers in addition to the functional aspects, the non-functional ones like user preferences, QoS of services, devices capabilities, etc. For the monitoring aspect, the both categories detects the environment changes by using sensors of the underlying environment (Sousa and Garlan, 2002) (Floch et al., 2006). Their focus is to capture the appearance/disappearance of components or devices. Few of work consider the changes of user preferences as an adaptation context (Sousa and Garlan, 2002). Most of the cited adaptation approaches corresponds either to the parametrization or to the behavioural adaptation that aims to reselect new components. In these cases, the adaptation is achieved by calculating new reconfigurations or using predefined plans to reconfigure their compositions (Floch et al., 2006) (Becker et al., 2004a).

2.5 Conclusion

In this chapter, we provided an overview of the existing approaches and middleware for the continuity of executions of applications in pervasive environments, which is composed of three complementary aspects: composition, monitoring and adaptation. Services composition aims to realize applications despite the highly heterogeneity of devices in pervasive environments, whereas, monitoring the environments is essential in highly dynamic environments due to the mobility of users and devices. However, services adaptation tries to overcome the captured failures and problems to ensure a continuous execution.

We have divided the cited work in two main categories depending upon how the application is described. First category consists of approaches dealing with the continuity of execution of applications having abstract descriptions. However, the second category cites work that tackle the composition, monitoring and adaptation issues of applications composed of concrete components.

In the following chapters of this thesis we present our contributions for these different aspects for the continuity of execution by considering many of the limitations of the

existing approaches that we identified in this chapter. We consider that a user task is defined as a composite of abstract components. First, we present our resolution approach to fulfil the composition of the task's services in Chapter 3. The monitoring will be detailed in Chapter 4. Finally task adaptation will be treated in Chapter 5.

Chapter 3

User Tasks Resolution

3.1 Introduction

Using Service-Oriented Architecture (SOA), a user task can be defined as an assembly of abstract components (i.e. services), requiring services from and providing services to each other. To achieve the task's execution, it has to be resolved into concrete components, which involves automatic matching and selection of components across various devices in pervasive environments. The implementation of these services can be found by looking up concrete and deployed components in devices of the environment. A service is matched with a concrete component if their interfaces match. A user task is said to be resolved if for all of its services, we find matching concrete components implementing these services.

In this chapter, we detail our contribution for resolution of abstract user tasks in pervasive environments. Our goal is to allow for each service of a user task, the selection of the best device and component for its execution. First, we examine some existing proposals for task' s resolution and list their limitations in Section 3.2. Then in Section 3.3, we introduce non-functional constraints which are considered for the selection of devices

and components. Section 3.4 describes our approach for the devices and components selection which are explained through an example scenario in Section 3.5. Finally, Section 5.6 concludes this chapter with an overview of our resolution contribution.

3.2 Related Work

In this section, we detail some of the existing tasks resolutions approaches as well as their limitations.

3.2.1 SeGSeC

In (Fujii and Suda, 2005) (Fujii and Suda, 2009), authors propose an architecture to dynamically compose the requested services, which are expressed in a natural language. Their proposal consists of realizing an application through combining distributed components based on their semantics and the user preferences.

To satisfy the composition of semantic components, the architecture consists of three sub-systems: Component Service Model with Semantic (CoSMoS), Component Runtime Environment (CoRE), and Semantic Graph based Service Composition (SeGSeC). CoSMoS is an abstract component model designed to model the functions (i.e., inputs, outputs or properties), the semantics (i.e., what each input, output and property semantically corresponds to), and contexts (i.e., their location, capability) of components. Each input (and output) of an operation is modeled as a component, representing that the operation accepts (or generates) another component as its input (or output). Similarly, a property of a component is also modeled as a component, representing that it can be retrieved as another component. CoSMoS models the information about a component as a semantic graph that consists of labeled nodes and links.

The services required for a composition are discovered through a middleware CoRE and composed based on their semantics and user preferences through a Semantic Graph-Based Service Composition (SeGSeC) mechanism. SeGSeC consists of four modules: RequestAnalyzer, ServiceComposer, SemanticsAnalyzer, and ServicePerformer as depicted in Figure 3.1.

The RequestAnalyzer parses the user request that is expressed in a natural language into a CoSMoS semantic graph representation. Then it passes it to a ServiceComposer. In order to create an execution path, the ServiceComposer seeks services from directories that can be supplied as the inputs of the operations. After discovering the services, it computes all possible combinations of the services that are sorted based on their similarity values to the user request (number of common node in the combination). After that, the ServiceComposer gives the execution path (in the form of semantic graph) and the user request (which is also a semantic graph) to SemanticsAnalyzer to check whether the user request is satisfied by the semantics of the execution path.

The SemanticsAnalyzer applies the semantic matching rules onto the execution path in order to derive the semantics of the path. Then, it notifies the ServiceComposer that the given execution path satisfies the user request. In this case, ServiceComposer passes the execution path to ServicePerformer to execute the given execution path by accessing the services, i.e., by invoking operations and retrieving properties of services in the specified order, through specific access interface.

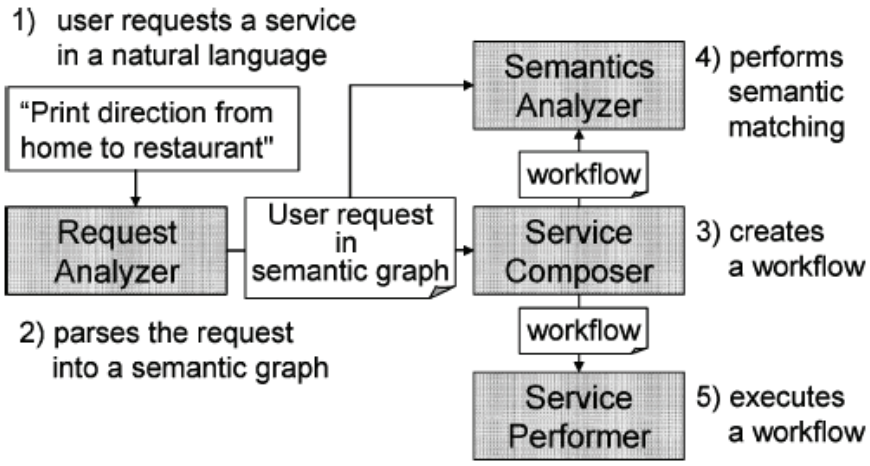


Figure 3.1: Architecture of SeGSeC (Fujii and Suda, 2005)

Thus, based on the semantic descriptions of services and the user preferences, SeGSeC composes services to satisfy a user request. However, this approach does not consider the services requirements, which have also an important influence on the services execution. Moreover, SeGSeC select arbitrarily discovered services to build the execution path without considering the case of similar services.

3.2.2 COCOA

COCOA is a COnversation-based service COmposition middlewAre (Ben Mokhtar et al., 2007) that allows a dynamic realization of user tasks from networked services available in the pervasive computing environment.

COCOA uses COCOA-L, which is an OWL-S based language for the specification of user tasks in pervasive environments. A service of a user task provides a set of capabilities. A service capability corresponds to either a primitive operation of the service or a process composing a number of operations (called conversation). To enable the automated reasoning about conversion behavior, they specify these capabilities as a finite state automata.

COCOA-L allows also the specification of required QoS properties. A user task has two kinds of required QoS properties that allow the description of both quantitative (e.g., service latency) and qualitative (e.g., CPU scheduling mechanism) non-functional properties. First, the QoS properties specified at the level of capabilities, express local QoS requirements. These QoS will be considered during services composition in order to be satisfied by individual advertised services. Second, the QoS properties specified at the level of the whole task, express the global QoS requirements.

The middleware performs services composition in two steps. The first step is to match each capability involved in the task's description against capabilities of the networked services. This is achieved by the COCOA Service Discovery algorithm (COCOA-SD). The second step is to compose the task's conversation using conversations of the selected

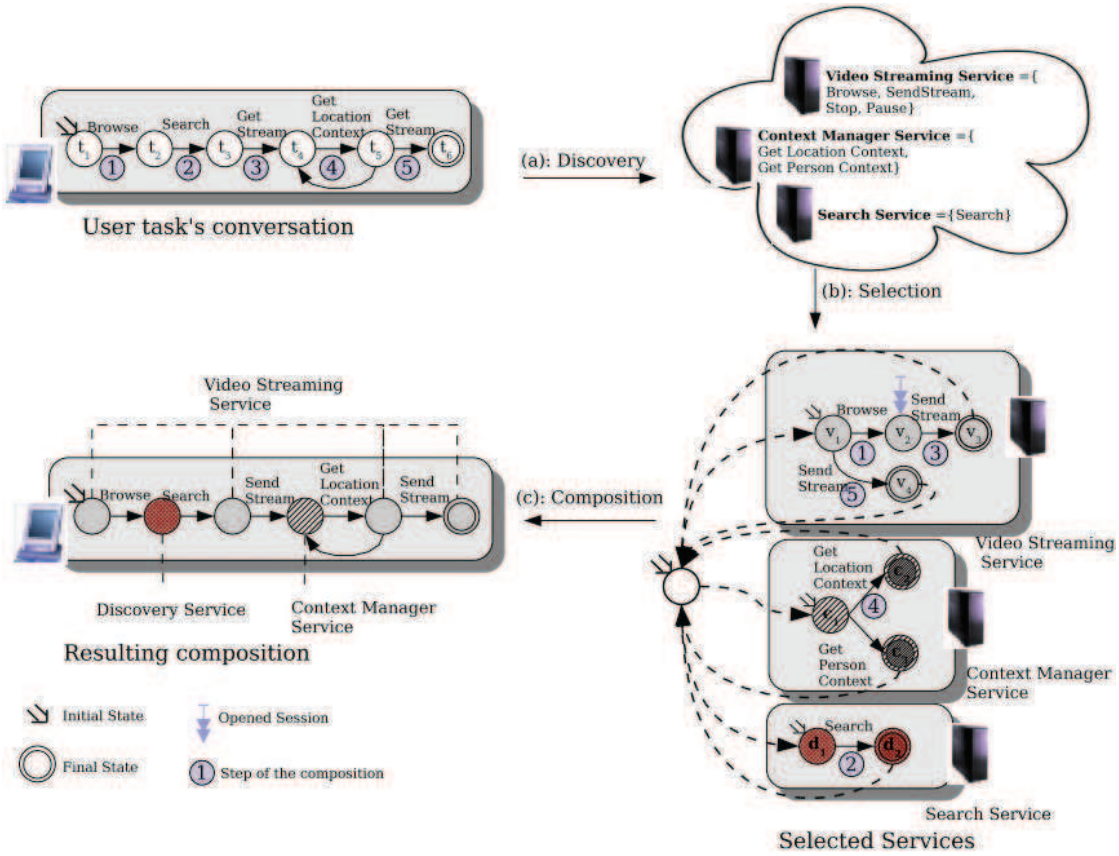


Figure 3.2: Service composition in COCOA (Ben Mokhtar et al., 2007)

services. This is achieved by the COCOA Conversation Integration algorithm (COCOA-CI).

COCOA-SD allows finding in the pervasive environment service advertised capabilities that match service requested capabilities for the realization of user tasks. COCOA-SD is decomposed into service matching and service selection. Service matching allows identifying services that provide semantically equivalent capabilities with those of the user task's conversation. Service selection allows identifying services that offer semantically equivalent capabilities to the capabilities of the user task and potentially useful for the composition by checking the local QoS requirements.

COCOA-CI integrates the conversations of the services selected by COCOA-SD to realize the conversation of the target user task. The global QoS requirements has to be satisfied by the resulting service composition. Therefore COCOA-CI checks the aggregation of QoS properties coming from the multiple advertised capabilities to be integrated. Then, it integrates all the automata of selected services in a global one.

Figure 3.2 shows the service composition process in COCOA. As it can be seen, the first step is a semantic matching of interfaces, that leads to the selection of the set of services that may be useful during the composition. Then, COCOA performs a conversation matching starting from the set of previously selected services, thus, obtaining a conversation composition that behaves as the task's conversation. The matching is based

on a mapping of OWL-S conversations to finite state automata. This mapping facilitates the conversation composition process, as it transforms this problem to an automaton equivalence issue. Once the list of sub-automata that behaves like the task automaton is produced, a last step consists in checking whether the atomic conversation constraints have been respected in each sub-automaton. After rejecting those sub-automata that don't verify the atomic conversation constraints, COCOA selects one of the remainders after checking their QoS. Using the sub-automaton that has been selected, an executable description of the user task that includes references to existing environment's services is generated, and sent to the Service Discovery and Invocation that executes this description by invoking the appropriate service operations.

COCOA middleware is limited on a semantic composition of user task services. It supports QoS for the services selection. However, the services composition does not consider neither user preference nor services requirements which may constrain the selection. Moreover, the COCOA framework permits the arbitrary selection of the resulting service composition as they all conform to the target user task.

3.2.3 Task Resolution Using Three-Phase Protocol

In (Mukhtar et al., 2011) (Mukhtar et al., 2009), authors propose an approach for the resolution of abstract user task by considering in addition to the functional aspects of the task, the user preferences, the services requirements and the devices capabilities. The resolution is carried out by a Task Composer which is provided by the user device where the user task is initiated.

The Task Composer first determines the abstract services, their interfaces, and the capability requirements of all the services, from the user task specification. These services are used to create a graph representation of the user task. To create a network graph, the Task Composer forwards the graph of the task to a Service Discovery system and uses the three-phase task composition protocol as shown in Figure 3.3.

First, the Service Discovery system interrogates the environment for available devices and queries them for their capabilities. Each device sends back its capabilities to the requesting device. For each device, the Task Composer compares its capabilities with the services requirements and the user preferences for the task. Any device which does not meet the required device capabilities or user preferences, is eliminated from the graph (as shown by the shaded circles in the Network layer of Figure 3.3). Then, devices will be valued for the overall task by considering the user preferences and devices capabilities and ranked from high to low values.

The second phase is the graph aggregation. At this level, the Service Discovery sends the abstract services to the selected devices starting from high to low value. Each device sends back a description of those components which match with at least one of the abstract services of the task. The Task Composer then combines the various components, obtained from all devices, into a single component graph.

Finally, the task composer will determine the final set of components that represent the concrete user task graph. If the device host more than one component offering similar functional interfaces, the task composer will select arbitrarily a component for the execution of the task's service.

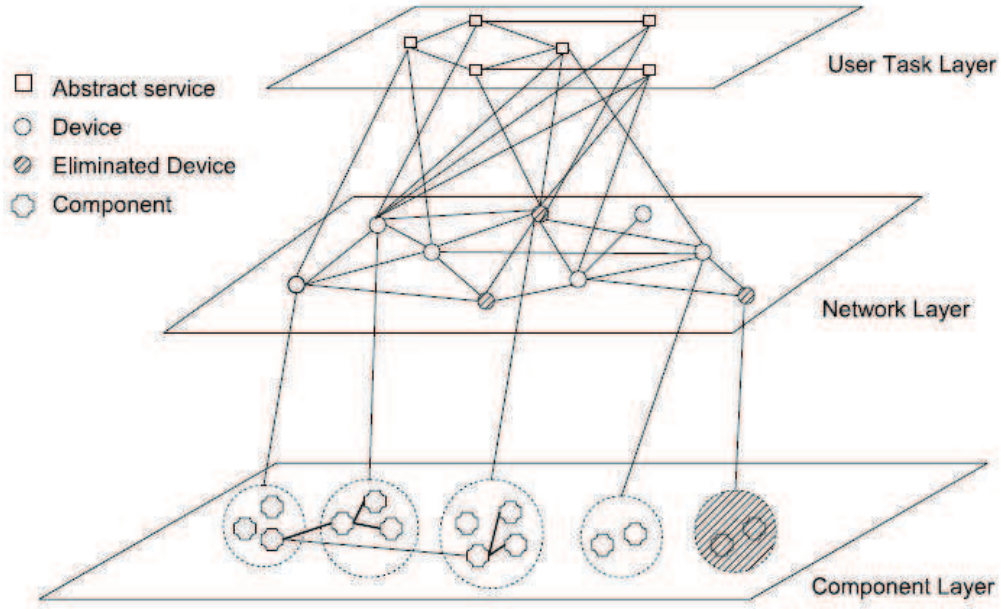


Figure 3.3: Task resolution at three different layers (Mukhtar et al., 2011)

While this approach responds to the most important features for a resolution of abstract user tasks as we intend to ensure, it presents some limitations. First, the selection of devices is done by valuing them for the overall task regarding the user preferences and their capabilities. However, this selection may prevent devices with lower value to not be selected even if they respond better to services requirements. Second, the selection of components is done arbitrarily. However, components may offer similar functional interfaces, but may differ from each other in terms of the needs towards the devices capabilities. Thus, there is a need to provide a mechanism for the selection of the most convenient component.

In view of all these identified problems in the above cited works, we now propose our own approach that overcomes these problems. This approach is based on the device selection approach presented in (Mukhtar et al., 2009) that is improved to ensure the selection of the best device and component for each service of the task.

3.3 Selection Constraints

Resolving an abstract task corresponds to the selection of concrete components that best match with its services across various devices provided by the execution environment. In order to select the most promising components for the preferred devices for the user, we require considering some non-functional constraints like devices capabilities, user preferences, services requirements and components preferences. In the following, we describe each of these non-functional constraints.

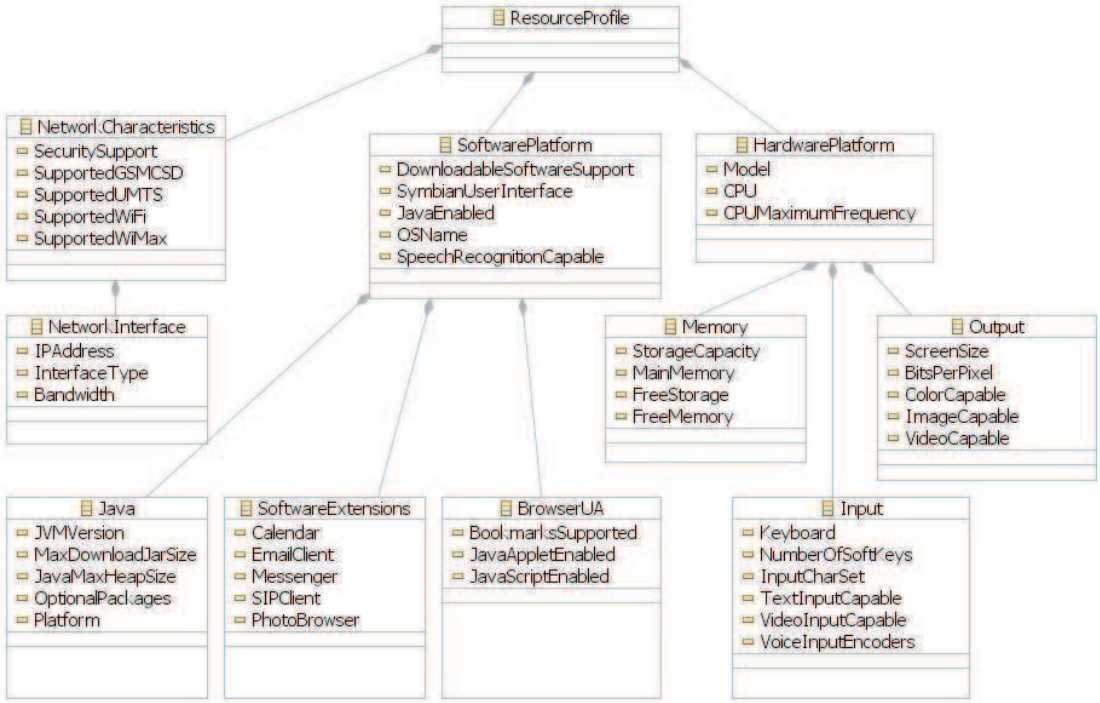


Figure 3.4: The extended CC/PP model

3.3.1 Device Capabilities (DC)

Pervasive environments consists of a variety of devices that may have a wide range of different characteristics: hardware, software, communication capabilities, etc.

All these aspects (i.e., software, hardware and network characteristics) represent the devices capabilities. As defined in RFC 2703 (Klyne, 1999), a *capability* is an attribute of a sender or receiver (often the receiver) which indicates an ability to generate or process a particular type of message content. It is possible to quantize a concrete capability, e.g., memory, CPU, bandwidth, etc.

Applications designed for pervasive environments do not consider the availability of particular software/hardware capabilities at the time of their conception. It is only at the time of execution of such applications, based on the availability of devices and the available components on them, that an application must know about the characteristics of the devices in the environment in order to determine devices for the selection of components. However, due to heterogeneity of devices, it is hard to assume in advance about the characteristics of individual devices.

In the literature, there have been a number of device description mechanisms like UAProf (Open Mobile Alliance (OMA), 2001), WURFL (Passani and Trasatti,), etc. CC/PP (Composite Capability/Preference Profiles) being an adopted standard, is the obvious choice for device description in now-a-days (Kiss, 2007) (Klyne et al., 2004). However, CC/PP descriptions are limited to only two-level hierarchy, consisting of components, and attributes attached to each component. Therefore, (Mukhtar et al., 2008a) proposes an extension to CC/PP that classifies a device capabilities into hardware, soft-

ware and network categories as shown in Figure 3.4. We use this extension to model the devices capabilities.

The hardware capabilities include the input and output of the device. While the software capabilities specifies software application installed on the device, the network capabilities represent a network access method or protocol, or a non-functional aspect such as security or payment. All of these capabilities are modeled in (Mukhtar et al., 2008b)(Mukhtar et al., 2008a) either as boolean or literal type, which can represent the device ability for a specific device capability. We denote by 1 that the device is able to provide a specific capability and 0 if not.

3.3.2 Service Requirements (SR)

Services in the user task may also describe their requirements for devices capabilities so that they can be executed only on devices fulfilling the required capabilities. These requirements may be specified by the task designer and they tailor the selection of devices to a task resolution.

We adopt the proposed model in (Mukhtar et al., 2009) to describe the services requirements as abstract resources requirements. The availability or absence of a capability on a device is noted by `true` and `false`, which can be represented by 1 and 0, respectively.

Assume for example a `TextService` specifying a `Hardware.InputCapable` as a service requirement. This abstract requirement implies that the device must be capable of accepting input from the user in order to use the service. Depending upon the interactivity model used by the device, text input can be accepted by a device in several ways: touch-screen, keypad, soft-keypad, keyboard, and mouse (using virtual, on-screen keyboard by clicking), etc. Thus, instead of specifying one of these several types as a required capability, the service may specify the capability abstractly using the `Hardware.InputCapable`. Thus, the device executing the service must be able to accept input resource from the Hardware category, otherwise the service will not work properly.

For a boolean capability `Hardware.TextInputCapable`, the possible value can either be `true` or `false` for a device. When a device is capable of accepting text as input, the capability will be presented as `Hardware.TextInputCapable=true`, which will be translated by the value of 1.

Hence, by introducing service requirements, we can reduce the solution space largely by decreasing the number of candidate devices. If any of the required capability's value is false, the service cannot be executed on that device. Hence, this later is eliminated as the service requirements are not satisfied there.

3.3.3 User Preferences (UP)

A user can specify his preferences and dislikes towards the devices capabilities in order to select the device that meets best his preferences for a particular task.

We adopt the user preference model as proposed in (Mukhtar et al., 2009), to represent the user preferences as real number values ranging between $(-1.0, 1.0)$. Using 1.0 value represents a very important capability, thus, a device should provide that capability else it will be eliminated. The -1.0 value represents users dislike to avoid using a device with such a capability, whereas, the 0.0 value represents a do not care condition, i.e., the availability or unavailability of such a capability is not important for the user.

A user may specify their preferences for device capabilities and resources with the help of a GUI provided on the device such as by clicking certain choices, by enabling/disabling certain options, or by specifying a preference value directly. Depending upon the value, the preference dictates the approval or dislike of the user for a particular capability of the device.

Based on the user preferences, the user task will be executed on a set of devices for which maximum of their preferences are satisfied.

3.3.4 Components Preferences (CP)

Compared to user preferences, a component may have preferences towards the devices capabilities, implying that the device should provide specific capabilities for which maximum of their preferences are satisfied. We proposed to model them using real values ranging between (0.0 , 1.0) (Ben Lahmar et al., 2011b). The 1.0 value represents a very important capability that a device should provide. If a component does not have any preference, we assume that it does not care, and its value is 0.

By considering these preferences, the user task will be executed by using concrete components for which maximum of their preferences are satisfied.

3.4 Principle of the Task Resolution

Resolving a user task corresponds to the selection of concrete components that best match with its services across various devices provided by the execution environment.

In (Mukhtar et al., 2009), authors propose an algorithm that evaluate devices regarding the overall task by considering the devices capabilities and the user preferences in order to select the device with the highest value. The limitations of this approach are on one hand, that the device value does not consider the services requirements towards the devices capabilities. Thus, it will prevent devices with lower values to be selected even if they respond better to services requirements. On the other hand, their algorithm does not provide any selection method for a suitable component if the selected device has more than one concrete component matching the same service.

In (Ben Lahmar et al., 2011b) (Ben Lahmar et al., 2012), we proposed an algorithm for the selection of the best device for each service of a user task, which improves the device selection algorithm in (Mukhtar et al., 2009) by considering in addition to the user preferences and the devices capabilities, the services requirements. We also proposed a new algorithm for the selection of the best component for each service of the task by considering the devices capabilities and components preferences.

In the following, we detail the both algorithms that are based on formulas presented in (Mukhtar et al., 2009) to evaluate the devices values.

3.4.1 Device Selection

To select the devices used for the user task's execution, we calculate a device value (DV) for each of them as following. Given a capability c_i of a device and the user preference or dislike value v_i for c_i, a weighted capability ω_i is calculated as:

$$\omega_i = \begin{cases} v_i & \text{if } c_i = 1, \quad -1 \leq v_i \leq 1 \\ -v_i & \text{if } c_i = 0, \quad -1 \leq v_i \leq 1 \end{cases} \tag{3.1}$$

Thus, a weighted capability is numerically equivalent to the user preference, but depending upon the presence or absence of a capability, it will take negative or positive value. This applies to most of the device's capabilities related to input methods, memory, CPU, etc., that can be represented by boolean or literal type (Mukhtar et al., 2008a) (Kiss, 2007). However, if we consider the screen size, it is a two dimensional capability and cannot be modeled like other capabilities. Let $R_u = \{w_u, h_u\}$ and $R_d = \{w_d, h_d\}$ represent respectively the screen resolution specified by the user and the screen resolution available on the device, where w and h represent the width and height dimensions of the screen. We used the Match method (formula 3.2) to return a value representing the matching degree between R_u and R_d of the device.

$$Match(R_u, R_d) = \frac{\frac{Min(w_u,w_d)}{Max(w_u,w_d)} + \frac{Min(h_u,h_d)}{Max(h_u,h_d)}}{2} \tag{3.2}$$

Using this equation, only the exact match will return the value of 1, while both larger and smaller screen sizes will return smaller values.

We used these formulas to calculate the device value (DV) as the following:

$$CalculateDV = \sum \omega_i + Match(R_u, R_d) + \frac{P_s}{P_t} - \frac{D_s}{D_t} \tag{3.3}$$

The P_s/P_t ratio is used to calculate the number of preferred or liked capabilities (P_s) satisfied by the device among the total number of preferred capabilities (P_t). We also consider the number of disliked capabilities available on a device. The D_s denotes the number of disliked capabilities present on a device, while D_t denotes the total number of dislikes specified by the user. A device with relatively high number of disliked capabilities will result in relatively lower value.

Using the formula 3.3, we denote by CalculateDV(DC, UP) to calculate value of a device for all of its capabilities (DC) and regarding the user preferences (UP). However, if we calculate a value of a device for a specific service by considering only the device capabilities (DC) and the user preferences (UP) related to its service requirements (SR), we denote it as CalculateDV(SR, DC, UP).

Algorithm 3.1 shows the main feature of the device selection process (Ben Lahmar et al., 2012). A user task consists of a list of services to resolve in a set of components provided by devices of the execution environment. We consider the fitness constraint proposed in (Mukhtar et al., 2009) to indicate that a service is executable on a device using a fit() method (line 6) that returns true if the service is executable on the device, or false otherwise. Indeed, each service may specify one or more capability requirements, which must be satisfied in order for the service to be executed on the device. If any of the required capability's value is *false*, the service cannot be executed on that device.

Moreover, two or more services may specify their colocation dependency on each other, which means that they must be executed on the same device (Mukhtar et al., 2009). A colocation request may arise due to intrinsic inter-service dependency or may be the result of a user preference. For this goal, the algorithm checks the fitness of the

Algorithm 3.1 DeviceSelection(Task t)

```
1:  DevicesList contains list of devices;
2:  DevicesTable: each row of this table is an ordered devices list associated with a service
    of the user task
3:  for each service s_i ∈ t do
4:    for each device d in DevicesList do
5:      if fit(s_i, d) and fit(colocation(s_i), d) then
6:        for each service s' in {s_i, colocation(s_i)} do
7:          if Requirement(s') ≠ ∅ then
8:            DV = DV + CalculateDV(SR, DC, UP)
9:          else
10:           DV = DV + CalculateDV(DC, UP)
11:         end if
12:       end for
13:       store the couple < d, DV > in DevicesTable(s_i) the row of the DevicesTable
          associated to s_i
14:     end if
15:   end for
16:   sort DevicesTable(s_i) from high to low values
17:   select DevicesTable(s_i)[1].d as the device has the highest value for s_i and
      colocation(s_i)
18: end for
```

colocation services (i.e. colocation() method in line 5) to determine if they fit in the same device as the task's service.

To calculate the devices values for the service and its colocation, the algorithm uses the formula 3.3. If the service specifies some requirements (Requirement method in line 7), the device value is calculated using CalculateDV(SR, DC, UP) (line 8). Otherwise, the device value is calculated using CalculateDV(DC, UP) (line 10). After that, it sums the device values calculated for the execution of the service and its colocations. The device with the highest value will be selected to fit a service and its colocations in.

Algorithm 3.1 improves the device selection algorithm in (Mukhtar et al., 2009), in the way that it evaluates each device for each service of the task rather than for the whole task. This improvement reduces the complexity of calculating devices values for each service of the task by considering only the capabilities and the user preferences that are related to its requirements.

3.4.2 Component Selection

Some selected devices may provide more than one component implementing the same service. In (Mukhtar et al., 2009), authors propose to select arbitrarily a component matching with the service. These components may offer similar functional interfaces, but they may differ from each other in terms of the capabilities of the devices. Thus, there is a need to select a component that best matches the service's description. For this, we proposed an Algorithm 3.2 that allows the selection of the convenient component for each service of the user task (Ben Lahmar et al., 2011b) (Ben Lahmar et al., 2012).

A component value (CV) is calculated using the following formula:

$$CalculateCV = \sum \omega_i + \frac{P_s}{P_t} \tag{3.4}$$

The ω is a weighted capability of a device that depends on the component's preferences. It is calculated using formula 3.1 by considering the components preferences instead of user preferences. The P_s/P_t corresponds to the number of preferred capabilities satisfied (P_s) among the total number of preferred capabilities (P_t) of the component.

Algorithm 3.2 ComponentSelection(Service s, Device d)

```
1: DeviceComponents contains components deployed on d
2: ComponentsTable: each row of this table is an ordered components list matching with
   a service of the user task
3: for each c in DeviceComponents do
4:   if s matches c then
5:     CV=calculateCV(CP,DC)
6:     store the couple < c, CV > in ComponentsTable(s) the row of the Compo-
       nentsTable associated to s
7:   end if
8: end for
9: sort ComponentsTable(s) from high to low value
10: select ComponentsTable(s)[1].c as the component has the highest value for s
```

Algorithm 3.2 uses the formula 3.4 to calculate components' values (line 5) by considering their preferences (CP) towards the device capabilities (DC). Then, the components values are sorted in order to select the component with the highest value. To achieve the task resolution, this algorithm will be executed in each selected device to choose the convenient component for each service.

3.5 Example Scenario

Referring back to the video player task in the chapter 1, which is represented by an assembly of three services: a VideoDecoder, a DisplayVideo and a Controller services. The Controller service sends a command to the VideoDecoder service to decode a stored video into appropriate format. Once the video is decoded, it is passed to the DisplayVideo service to play it. Assume that the DisplayVideo service should be collocated with the VideoDecoder for a quick transfer of frames. Thus, they should be executed in the same device.

For the task execution, the services should be resolved into concrete components available in the execution environment. Consider that this later consists of a Smartphone (SP) and Flat-screen (FS) devices and that each device provides concrete components matching with the task's services in order to valid their fitness. Table 3.1 shows some capabilities of the two devices and their corresponding user preferences.

Moreover, services may express their requirements towards the devices capabilities. For example in table 3.2, the DisplayVideo service requires an output VideoCapable

Table 3.1: Device capabilities and user preferences

Capabilities	SP	FS	User
Resource.Software.VideoPlayer=VLC	1	0	0.1
Resource.Hardware.Output.VideoCapable.Screen.Width	320	1920	1920
Resource.Hardware.Output.VideoCapable.Screen.Height	240	1200	1200
Resource.Hardware.Output.SoundCapable.InternalSpeaker	1	1	0.2
Resource.Hardware.Output.SoundCapable.ExternalSpeaker	1	1	1.0
Resource.Hardware.Input.Keyboard	1	0	0.1
Resource.Hardware.Input.TouchScreen	1	0	0.3
Resource.Hardware.Memory.MainMemory	1	1	0.0
Resource.Hardware.Memory.Disk	1	0	0.0

and SoundCapable capabilities to achieve its execution, whereas, the Controller service requires an input capability to command the video player task.

Table 3.2: Requirements of the video player services

Service	Service Requirements
Controller	Resource.Hardware.Input
DisplayVideo	Resource.Hardware.Output.VideoCapable
	Resource.Hardware.Output.SoundCapable

To resolve the video player task, there is a need to select for each service of the task the best device that has the highest value. In the following, we compare the results of our Algorithm with the device selection Algorithm in (Mukhtar et al., 2009) to prove the convenience of our approach.

Table 3.3: Devices values using Device Selection Algorithm in (Mukhtar et al., 2009)

SP	FS	Selected Device
2.78	2.3	SP

Table 3.3 shows the devices values for the overall task regarding the devices capabilities and the user preferences as described in (Mukhtar et al., 2009). The SP has the highest value, thus, it will be selected to resolve the video player task. However, this selection does not satisfy the user preference which is displaying video in bigger screen.

We have evaluated the devices for each service of the video player task by considering only the capabilities related to each service requirements and their corresponding user

Table 3.4: Devices values for Controller and DisplayVideo services

Service	SP	FS	Selected device
Controller	1.4	-0,4	SP
DisplayVideo & VideoDecoder	5.16	5.5	FS

preferences. Table 3.4 shows the devices values as a result of Algorithm 3.1. The Controller service will be executed in SP device, while the DisplayVideo and VideoDecoder, which are two colocated services, will be executed in FS device. As it can be see, our algorithm satisfies best the user preference (i.e. bigger screen size) compared to algorithm in (Mukhtar et al., 2009).

To achieve the service resolution, there is a need to select the suitable components in the selected devices. This is done by each selected device by using the Algorithm 3.2 that allows the selection of the convenient component for each service of the video player task by considering the preferences of components regarding the devices capabilities.

Table 3.5: Preferences of DisplayVideo components in the FS device

Preference	DisplayVideo1	DisplayVideo2
Resource.Hardware.Memory.MainMemory	0.8	1.0
Resource.Hardware.Memory.Disk	0.2	0.0

In Table 3.5, we list the preferences of components provided by FS device. The DisplayVideo2 component requires a main memory to cache the video, whereas the DisplayVideo1 component is less interested in Disk memory than the Main one.

Table 3.6: Selection of component for DisplayVideo Service

DisplayVideo1	DisplayVideo2	Selected Component
1.1	2.0	DisplayVideo2

Table 3.6 presents the values of DisplayVideo components provided by FS device in order to select the suitable component for the DisplayVideo service. The DisplayVideo2 component has a higher value than the DisplayVideo1 component. Thus, it will be selected for the execution of the DisplayVideo service.

3.6 Conclusion

A user task can be defined as an assembly of abstract components (i.e. services), requiring services from and providing services to each other. To achieve the task's exe-

cution, it has to be resolved in concrete components, which involves automatic matching and selection of components across various devices.

Towards these challenges, we proposed in this chapter, algorithms that ensure for each service of a user task the selection of the best device and component for its execution. This is done by considering in addition to the functional aspects of a task, some non-functional ones like user preferences, devices capabilities, services requirements and components preferences. These non-functional features allow to refine the devices and components lists in order to select a suitable component in a convenient device, which have highest values, for each service of the user task.

Thus, using this approach, it is possible to resolve user tasks by combining deployed components provided by various devices that satisfies the services requirements and maximizes the user and components preferences regarding to devices capabilities. However, user tasks in pervasive environments are challenged by the dynamism of their execution environments due to users mobility, devices appearance and disappearance, etc. This may have an impact on the tasks' executions. Therefore, there is a need to detect these changes and then adapt tasks regarding the captured changes, which are the topics of the next chapters.

Chapter 4

Monitoring of Components

4.1 Introduction

Due to the heterogeneity and dynamicity of pervasive environments, an important aspect of user tasks is that their execution is very much dependent on their context. Therefore, modeling the behavior of a user task needs to satisfy not only the functional requirements in an effective way, but in order to provide better quality of service (QoS) for user satisfaction, it should also consider the current state of the environment in which the task is executing. In such situations, a component may depend on the changes of components' properties representing the environment or the user task. This gives a rise to the need to monitor these properties in order to allow the component to be aware about their changes.

In this chapter, we present our vision for the component model that allows components to express explicitly their dependencies to properties provided by components representing the execution environment or belonging to a user task. Moreover, we present

a monitoring mechanism to detect the changes of properties of local or remote components. If a component is not monitorable by default, we proposed some transformation mechanisms to render it monitorable whenever there is a need.

Before going into the details of our contribution to tackle the above mentioned challenges, we provide an overview of the existing related approaches as well as their limitations in Section 4.2. Then, we describe the principle of our monitoring approach in Section 4.3. Section 4.4 describes how local and remote components can be transformed to make them monitorable. In section 4.5, we present an example scenario through which we describe how the monitoring is handled. Finally, Section 4.6 concludes this chapter with an overview of our monitoring approach.

4.2 Existing Monitoring Approaches and Backrgound

The monitoring issue has been extensively studied in different contexts, notably in the area of software components that has become an accepted standard for building complex applications. In this section, we detail some of the existing related approaches as well as their limitations.

4.2.1 Observation Contracts

In (Beugnard et al., 2009), authors propose a process that makes functional aspects of components independent from observational ones in order to observe the execution context. This separation of concerns allows the advantage of changing observations without modifying the core part of components.

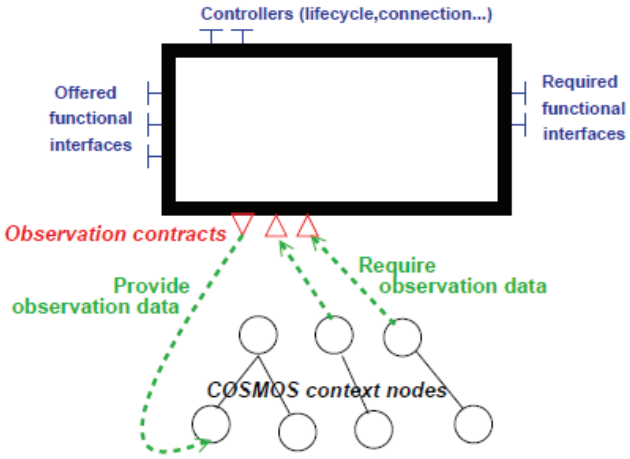


Figure 4.1: Component with observation contracts (Beugnard et al., 2009)

Towards this objective, they propose to define a set of predefined components dedicated to observation that can be attached to any functional component by complementing the Fractal component model (Bruneton et al., 2006) with observation contracts as drawn in Figure 4.1. The thick black box denotes the controller part of a component, while the interior of the box corresponds to its content part. Interfaces appearing on the left and

right sides depict server and client interfaces, respectively. Interfaces appearing at the top of the box represent reflexive control (extra-functional) interfaces such as the life-cycle controller, the binding controller, the attribute controller or the content controller interfaces. In the bottom of the component, observation contracts describes how a component and an observable nodes are associated. It defines if the observation data are provided or required by the component. In case of required observation data, it also defines the mode: observation or notification. For a notification, it defines what event triggers this notification.

COSMOS (Conan et al., 2007) component-based framework is used to describe the contextual situations to which a context-aware application is expected to react. These situations are decomposed into fine-grained units called context nodes. A context node is a context information modeled as a Fractal component. Communication between context nodes through the hierarchy may be bottom-up or top-down. The former case corresponds to notifications sent by context nodes to their parents, whereas the latter case corresponds to observations triggered by a parent node.

Hence, the monitoring contracts are used by components of applications to express their monitoring needs of contexts expressed as COSMOS context nodes. For this goal, these nodes should provide observation components through which they are monitored. Otherwise, the monitoring could not be handled. This will limit the monitoring to only defined observation components.

4.2.2 Pervasive and Social Bindings

In (Mélisson et al., 2010), authors propose pervasive binding to provide support for service discovery in applications based on SCA (Service Component Architecture) (Open SOA Collaboration, 2007). This binding is called UPnP binding since it is based on UPnP protocol (UPnP Forum, 2008). Towards this objective, they propose to integrate UPnP into FRASCATI platform (SCA runtime)(Seinturier et al., 2009) to support the remote call of a service that is advertised via the UPnP protocol.

They propose also to integrate social bindings into FRASCATI platform to allow asynchronous event exchange via Twitter. These bindings allow the notification of situations identified by the system. To do that, the SCA services defining Twitter bindings do not require to define specific interfaces. Instead, the services specify the representations and types of the events that can be retrieved from the Twitter accounts. This information will be used for marking the tweets with the event type that contains using Twitter tags. Thus, by encapsulating the event notifications as SCA social bindings, it is possible to support asynchronous event notification.

Hence, the FRASCATI platform was modified to supply spontaneous communications between SCA components using UPnP bindings and social binding. This limits the execution of applications only on the modified FRASCATI platform and it does not consider the existing open-source SCA runtimes (e.g. Newton, Tuscani, ...). Moreover, the social bindings is used to send notifications about some events related to Twitter account. Thus, they can not be used to notify components about events related to the changes of properties of other components.

4.2.3 Monitoring for SLA management

To ensure some Quality of Service (QoS) level in an application, (Ruz et al., 2010) (Ruz et al., 2011) propose to monitor Service Level Agreements (SLA) conditions, that are specified in contracts established between consumers and providers services, to cope with the evolving SLA.

Their proposal is to separate the concerns involved in an autonomic control loop (MAPE) (Computing et al., 2006) and implement those concerns or a set of them as separate components attached to each managed service, in order to provide a monitoring and management framework.

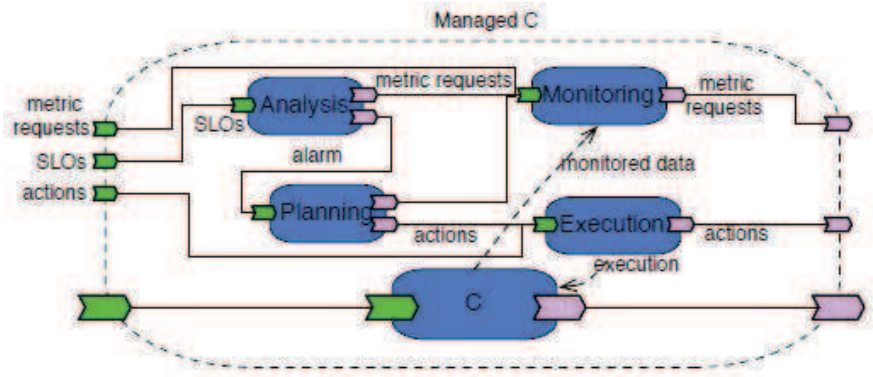


Figure 4.2: A service with its attached monitoring and management components (Ruz et al., 2011)

Figure 4.2 shows a service C that is extended with one component for each phase of the MAPE loop and converted into a Managed service C (dashed lines). The Monitoring component collects monitoring data from service C. They consider one Monitoring component attached to each monitored service, that collects information from it, and exposes an interface to obtain the computed metrics. Monitoring involves the collection of information from the target service (sensing), storage, filtering, and possibly processing (apply functions over the sensed values) of the data to obtain a set of metrics that is made available for other components. The implementation of the Monitoring component should support the communication with sensors provided by the service C or, alternatively, it must include specific sensors to monitor it. This way the Monitoring component is effectively attached to the service, which becomes a monitored service from the framework point of view. The monitored metrics may be exposed in a pull mode, where the interested component asks explicitly for a value, or in a push mode, where the component can ask for a metric and receive a periodic update each time that the value changes.

The Analysis component checks the compliance to a previously defined SLA. An SLA is defined as a set of simpler Service Level Objectives (SLOs), which must be verified at runtime. In any case a parsing component specialized in reading this description and creating the desired SLO must be used. As input, the Analysis receives a set of conditions (SLOs) to monitor, expressed in a predefined language. The Analysis checks the compliance of all the stored SLOs according to the metrics obtained from the Monitoring component. The Analysis checks if the SLA is being fulfilled, and if not, it sends an alarm notification to the Planning component.

The Planning component implements a strategy defined for reacting to an alarm notified by the SLA Analysis component, which is a sequence of actions. These actions are executed by the Execution component, which includes the specific means to make them effective over service C, completing the loop.

They prototyped a framework based on GCM (Grid Component Model) model (Baude et al., 2009), which enables large-scale grid/cloud deployment of components. The framework is implemented as a set of non functional components that can be added or removed at runtime to the membrane of any GCM component, which becomes a managed service of the application. For example, a service that does not need monitoring information extracted, does not need to have a Monitoring component and may only have an Execution component to modify some parameter of the service.

The limitation of their approach is that the management of a service is independent of other services. Thus, the framework components are dedicated for a single service and thus they cannot be used to monitor or to manage the changes of other services.

4.2.4 Publish/Subscribe Systems

The publish/subscribe interaction systems, also called event-based systems (Silva Filho and Redmiles, 2005), represent the earliest research work to cope with the monitoring challenge. Their objective is to provide subscribers with the ability to express their interest in an events, in order to be notified subsequently of any event, generated by a publisher, that matches their registered interest. In other terms, producers publish information and consumers subscribe to the information they want to receive. This is performed by means of subscriptions. The subscription denotes the act of expressing interest on some specific content, which can be performed in different ways such as: opening a communication channel between two or more parties, posting a filter expression, defining rules and queries on parts of this information content, becoming part of a group where this content is produced, and many other ways.

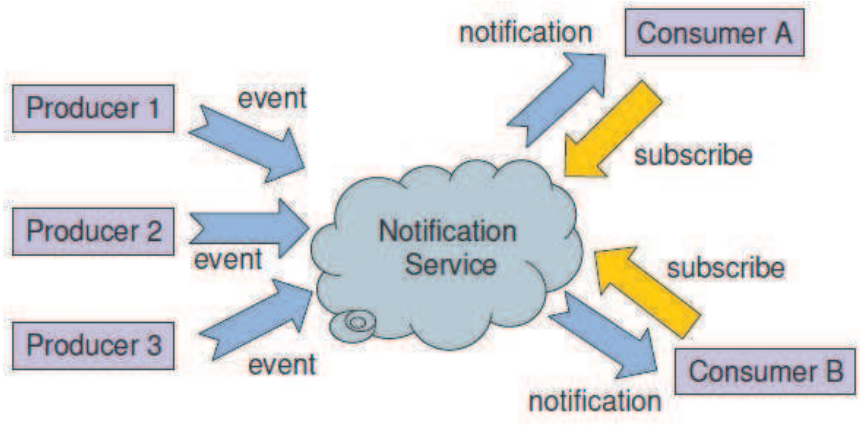


Figure 4.3: Basic components in a distributed publish/subscribe system (Silva Filho and Redmiles, 2005)

In Figure 4.3, we depict the different elements of a publish/subscribe system:

- Producers, also called the publishers, are the entities that produce information in the system. This information is called an event, which is published via the notification service.
- Consumers, also known as subscribers, are the entities that consume information. They express their interest in an event or a set of events by requesting one or more subscriptions to the notification service.
- Subscriptions represent the act of expressing interest in specific content by consumers. When a Consumer is no longer interested in this set of events, it cancels the subscription corresponding through the notification service.
- Events express a change of a component state. An event is expressed in the form of a message, transmitting the content or information on this event. The message can have different representations, such as plain text, tuples (attribute/value pairs) etc.
- The notification service is the main element of a publish/subscribe system and it is responsible for receiving subscriptions from consumers, and events coming from their producer. With these two sets of information, it efficiently performs the matching of subscriptions with their corresponding subset of events, routing the resulting eventsto the interested parties. Thus, it notifies consumers that an event is ready to be retrieved.

Moreover, publish/subscribe infrastructures supports different subscription languages. Subscribers are usually interested in particular events and not in all events. The different ways of specifying the events of interest have led to several subscription schemes.

(Eugster et al., 2003) classifies the subscription models into three main categories: topic-based model, content-based model and type-based model. The earliest publish/-subscribe model is based on the notion of topics or subjects. Participants can publish events and subscribe to individual topics, which are identified by keywords. Publish/subscribe systems based content allows the event system to use filters based on the content of the event. In other terms, events are not classified according to some predefined external criterion (e.g., topic name), but according to the properties of the events themselves. However, the subscription based type model uses concepts of object-oriented programming: the events are declared as objects belonging to a particular type that can encapsulate the attributes as well as methods (Baldoni and Virgillito, 2005).

We distinguish two main notification service architecture that are namely, centralized and distributed architectures (Carzaniga et al., 2001). The centralized architecture uses a single entity called the server (called also proxy or broker) to broadcast events between producers and consumers. However, a distributed event notification service is composed of interconnected servers, each one serving some subset of consumers and producers of the service. The distributed architecture is classified into three basic architectures: hierarchical client/server, acyclic peer-to-peer, and general peer-to-peer.

4.3 Monitoring Required Properties

4.3.1 Component Model

In an object-oriented paradigm, an object provides its services through a well-known interface, which specifies the functionality offered by the object. However, in component-

oriented paradigm, components may specify, in addition to their provided interfaces, their properties, through which they can be configured, and their dependencies on the offered services of other components using required interfaces.

As defined by (Szyperski, 2002), *"a software component is a unit of decomposition with contractually specified interfaces and explicit context dependencies only"*. Thus, a component exposes not only its services but it also specifies its dependencies. These dependencies represent not only the offered services of other components but also external properties. It is possible that a component may depend on the changes of other components' properties in order to adjust its internal state.

The ability to specify dependency for external properties has two important implications. First, it results in specification at relatively fine granularity thus helping the architects and designers in fine tuning the component's requirements. Second, this fine tuning helps in elaborating the contract between two components because the properties can be enriched with additional attributes that constrain the nature of the contract through appropriate policies. Thus, modeling the application behaviour needs to satisfy not only the functional requirements in an effective way, but also to consider external properties, for example, knowing the current state of the environment in which the application is executing in order to provide a better Quality of Services (QoS).

Most of the existing component models PCOM (Becker et al., 2004a), Fractal (Bruneton et al., 2006), OSGi (OSGI, 1999) and SCA (Open SOA Collaboration, 2007) allow specification of their dependencies for business services external to the component. However they do not allow specification of their dependency for external properties in explicit way. Some of which leave such issues to the underlying middleware, which provides a uniform Application Programming Interface (API) or a framework for this purpose. This means that the programmers and the designers have to rely on the functionality of the underlying middleware and such aspects need to be considered during application development life-cycle.

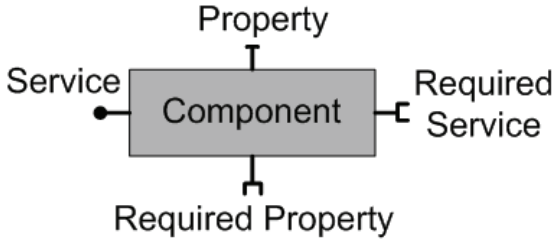


Figure 4.4: Component describing its required properties

Towards these issues, we proposed an extension to the concept of component to allow it expressing explicitly its dependencies to external properties in terms of required properties (Ben Lahmar et al., 2010b). Required properties are used to express the dependency of components to properties values of other components.

Figure 4.4 shows main characteristics of a component that provides a service through an interface and requires a service from other components. The component also exposes a property through which it can be configured. In addition, the component also specifies its dependency on a certain property. This required property, which appears at the bottom

of the component, will be satisfied if we can link this component with another component that offers the requested property, thus, solving the dependency.

4.3.2 Monitoring Specification

Using required properties allows a component to express its dependencies to external properties. It is possible that the component depends on the changes of these external properties. This requirement can be specified explicitly through its required properties. Thus, there is a need to a monitoring mechanism in order to allow it to be aware about their changes. In (Ben Lahmar et al., 2010b), we proposed a monitoring approach to permit to a component to express its needs to monitor the changes of its required properties.

Monitoring required properties correspond to detecting changes of offered properties of other components. Hence, each offered property can be monitored by other components that depend on its changes. The process consists in informing the interested component about the changes of required properties or notifying it on a regular way or for each variation.

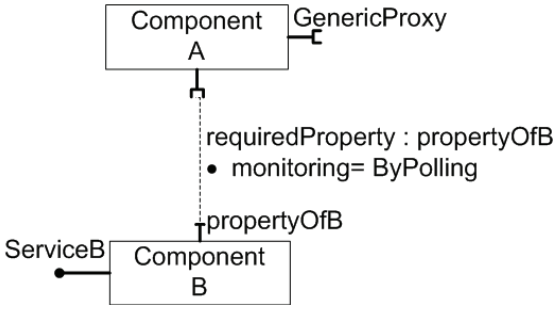


Figure 4.5: Specification of monitoring by polling

We distinguish two monitoring types: by polling and by subscription. Polling is the simpler way of monitoring, as it allows the observer to request the current state of an external property whenever there is a need. Figure 4.5 shows a component A expressing through its required property the need to monitor by polling an external property provided by a component B.

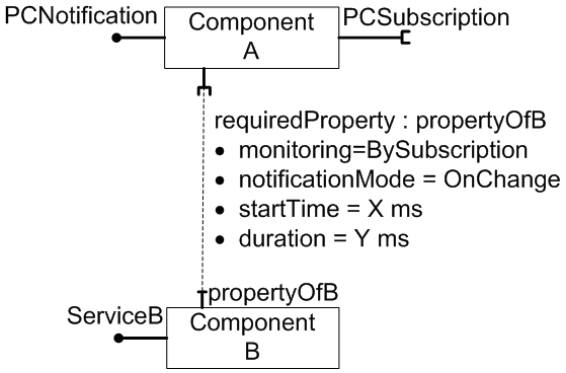


Figure 4.6: Specification of monitoring by subscription with notification mode OnChange

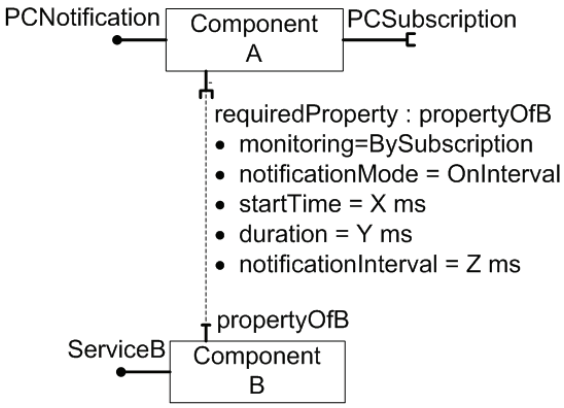


Figure 4.7: Monitoring by subscription with notification mode OnInterval

However, subscription allows an observing component to be notified about changes of monitored properties. There are two modes of monitoring by subscription: 1) subscription OnChange which specifies that the subscriber component is notified every time the value of the property changes; 2) subscription OnInterval which specifies that the subscriber component is to be notified after a specified time interval.

For the notification OnChange, a component may precise the starting time and the duration of notifications. For the notification OnInterval it must specify the notification interval value and may also precise the starting time and the duration of notifications. Figure 4.6 shows how the component A specifies its need to monitor a required property offered by the component B by subscription mode OnChange, whereas, Figure 4.7 shows its monitoring request by subscription OnInterval.

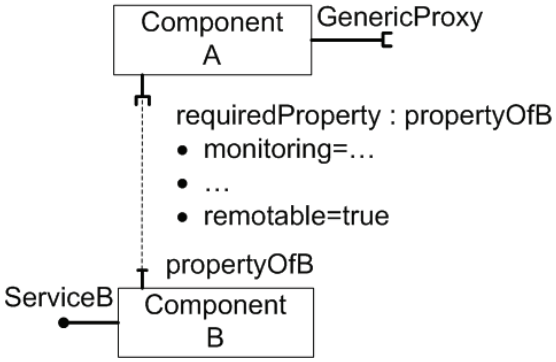
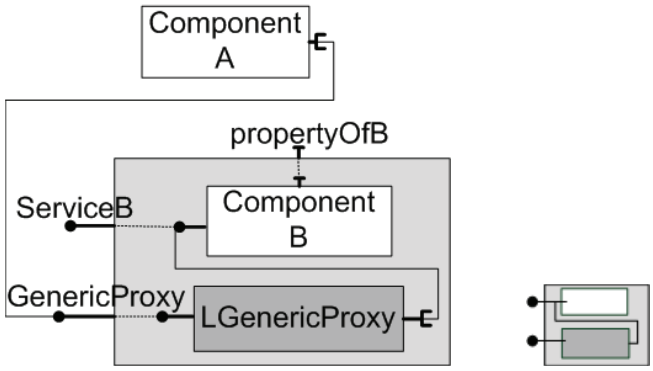


Figure 4.8: Specification of a remote monitoring need

To carry out a distributed application in a pervasive environment, some components may be interested in monitoring properties of remote components. Using required properties provides the ability to a component to specify that its required properties are offered by remote components. As the local case, the remote monitoring can be achieved by observing to the current state of the remote components, or by subscribing to their changes. Figure 4.8 shows how the component A specifies its need to monitor a property offered by the remote component B.



(a) (b)

Figure 4.9: Transformation of a local component for a Monitoring by polling

4.4 Transformation Mechanisms

The specification of required properties may contain a monitoring request by polling or by subscription of those properties. Therefore, the component maintaining the required properties, should allow the monitoring of its properties by offering a monitoring service and having a notification mechanism. However, some components may not define their offered properties as monitorable resources despite the components' requests.

Our objective is that given a user task, which was not conceived for dynamic environments with changing QoS, we would like to transform it in order to allow it to monitor the properties of components which depend on. The transformation may applied to some of its components available on the device locally resulting in local transformation or it may applied to a component in remote device, in which case it is known as remote transformation.

The transformation consists of creating a composite that encapsulates the considered component with predefined monitoring components (Belaïd et al., 2010). The new composite offers monitoring services in addition to the services of the considered component.

In the following, we present transformation mechanisms to render local as well as remote components monitorable in order to respond to monitoring needs of other components.

4.4.1 Local Monitoring

A component can be transformed in order to be monitored locally by polling or by subscription. In the following, we describe the transformation mechanism for each type of the monitoring.

Monitoring by Polling

A component may express its need to monitor by polling a required property provided by another component (see Figure 4.5). The monitoring by polling of a property can be

made by calling its getter method. However, the component that wishes to monitor a property of another component does not know a priori the type of this component. To complete the monitoring of any component from only the name and type of a property, the interested component often uses an appropriate interface that provides the method getPropertyValue(propertyName) to request the current state of a property.

```
public interface GenericProxy {
  Property[]  getProperties();
  Object  getPropertyValue(String propertyName);
  void  setPropertyValue(String propertyName, Object propertyValue);
  Object invoke(String methodName, Object[] params);
}
```

Figure 4.10: Description of the Generic Proxy interface

However, the component to monitor may not define its offered properties as monitorable by polling resources despite the request. So, we need to transform the component to make its properties to be monitorable by offering an appropriate interface of monitoring.

For this purpose, we defined a generic proxy service which can be applied to any component of a user task. A general purpose of the GenericProxy interface is to provide four generic methods that are described in Figure 4.10. Each implementation of this interface is associated with a component for which the first method getProperties() returns the list of the properties of the component, the getPropertyValue() returns the value of a property, the setPropertyValue() changes the value of a property and the invoke() method invokes a given method on the associated component and returns the result.

Thus, the transformation consists of a creation of a new composite that encapsulates the considered component with a LGenericProxy component, which provides a local implementation for the GenericProxy interface. Thus, when a LGenericProxy component is associated with a local component, it translates its method calls into calls of this latter. The new composite offers in addition to the service B, a GenericProxy service that allows a component to get the properties value of another component using its getProperties() method.

Moreover, the new created composite allows a component not only to observe the changes of its required properties, but also to reconfigure some of them if it is needed. This is ensured using the setPropertyValue() method of the GenericProxy interface.

In Figure 4.9(a), we show the transformation of the component B to render its property monitorable by polling by the component A, while Figure 4.9(b) is a symbol, we used to represent the transformation for monitoring by polling.

Monitoring by Subscription

There are two modes of monitoring by subscription: 1) OnChange and 2) OnInterval. For the the both modes of notification, the component B must offer a subscription service to the component A and in turn, the component A must subscribe to the component B specifying its need. When a change of the property happens, a notification is sent

from the component B to A. To provide the monitoring ability to the component B, we propose to transform it into a composite offering a monitoring interface. This is done by encapsulating it with MonitoringBySubscription and LGenericProxy components.

```
public interface PCSubscription {
  boolean subscribe(PCNotification listener, String propertyName);
  boolean renewSubscription(PCNotification listener, long duration);
  boolean unsubscribe(PCNotification listener, String[] names);
}
```

Figure 4.11: Description of the property changed subscription interface

```
public interface PCNotification {
  void notify(Object source, String propertyName, Object propertyValue);
}
```

Figure 4.12: Description of the property changed notification interface

The MonitoringBySubscription component implements a PCSubscription interface, as described in Figure 4.11, in order to allow a subscriber component to subscribe, to renew its subscription or to unsubscribe to property changed event. The MonitoringBySubscription component also implements a PCNotification interface, as depicted in Figure 4.12, in order to send notifications the interested components.

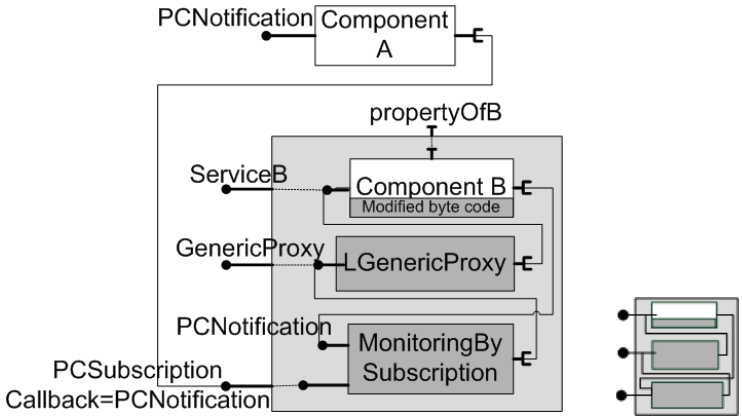
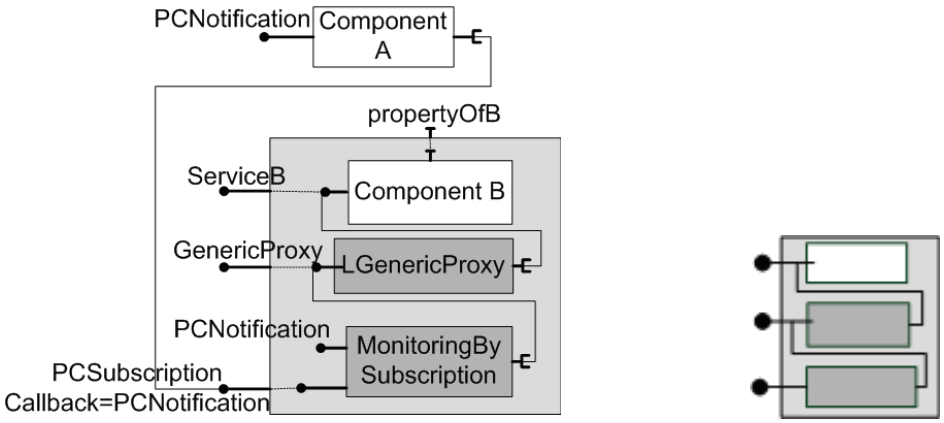


Figure 4.13: Monitoring by subscription with notification mode OnChange

When the notification mode is on change for a required property of B (Figure 4.13(a)), the MonitoringBySubscription component offers a (callback) service of notification PCNo-



(a) (b)

Figure 4.14: Monitoring by subscription with notification mode OnInterval

tification to the component B so that it can be notified of the changes of a required property and in turn inform all the subscribers of this change. To allow the component B to notify the MonitoringBySubscription for the change of its properties, the byte-code of the component B is modified at runtime by adding new instructions for notifications.

Thus, the created composite provides in addition to the GenericProxy service, a PCSubscription one, while the component A should specify through a required interface its need to PCSubscription service.

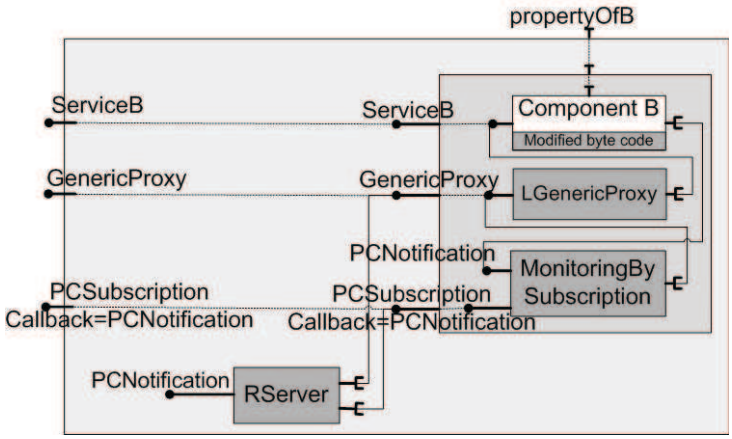
For the monitoring by subscription with notification mode on interval, as shown in the Figure 4.14(a), each time the MonitoringBySubscription component have to notify the subscriber (the component A), it gets (or monitor by polling) the value of the required property of the component B via the LGenericProxy component.

Figures 4.13(b) and 4.14(b) shows the symbols we used for denoting subscription on change and subscription on interval, respectively.

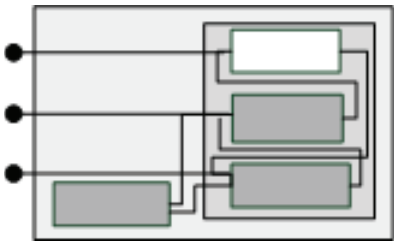
4.4.2 Remote Monitoring

As well as the local monitoring, a component can be transformed in order to render it monitorable to satisfy a remote component request (See Figure 4.8).

In case of the monitoring by subscription, whatever the notification mode, the monitored component (server side) must offer a remote subscription service over the network to the subscriber component (client side) and in turn, this later must subscribe to the remote server component specifying its need. However to provide a remote monitoring by polling the server component must offer a GenericProxy service and should also be reachable over the network as for the local case. We note that for the remote purposes there are two transformations: server-side and client-side (Belaïd et al., 2010) as following.



(a)



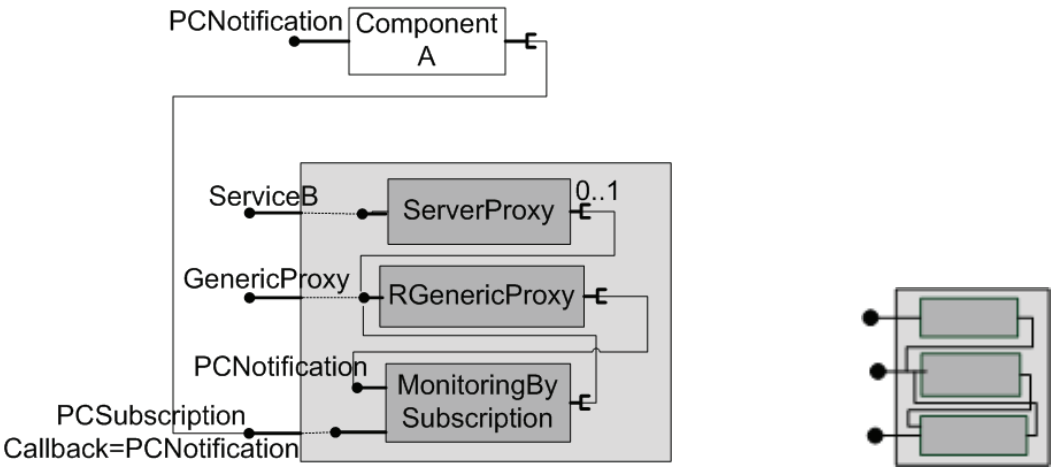
(b)

Figure 4.15: Remote monitoring: server side

Server-side transformation

To render a remote component B monitorable, the transformation consists first of encapsulating it in a composite (Figure 4.15(a)) such as defined for the local transformation for the monitoring by subscription. Then it adds a new RServer component that integrates the network communication aspects like remote call and event processing. The remote call of the RServer will be used by the service B, the GenericProxy service and PCSubsscription service to allow the remote subscriptions. However, the event processing of the RServer component is used to send notifications over network to the subscriber component once it is notified by MonitoringBySubscription component. The RServer component has two references: one for the subscription service offered by the MonitoringBySubscription component, which is used to subscribe to the changes of properties of the component B. The second reference is for the GenericProxy service provided by the LGenericProxy component, which is used for the monitoring by polling on behalf of the client component A.

We note that the newly defined composite provides the GenericProxy service and the PCSubscription service in addition to the services of the component B. Thereby it



(a) (b)

Figure 4.16: Remote monitoring: client side

can be used, for local monitoring. Figure 4.15(b) is a symbol denoting the server-side transformation.

Client-side transformation

A new composite is created in the client side to represent the remote component B. This composite provides the service B, a GenericProxy service and the PCSubscription one, as shown in Figure 4.16(a). The ServerProxy component is a byte-code generated component at runtime. Its implementation of the service B consists on forwarding its calls to the RGenericProxy component. The RGenericProxy component corresponds to a remote implementation of the GenericProxy interface. It forwards the calls of its methods over the network to the RServer component in the server side. The MonitoringBySubscription component is able to receive subscriptions requests from the component A that are forwarded to the RGenericProxy component for a transfer over the network to the RServer component..

When the RGenericProxy receives a notification from RServer component, it forwards it to the MonitoringBySubscription component that in turn notifies by callback the target component A. Figure 4.16(b) is a symbol, we used to represent the transformation in the client-side.

4.5 Example Scenario

Referring back the video player task presented in Chapter 1. Assume that the VideoDecoder component is dependent on the network signal property to decide the rate of frames transferred to the DisplayVideo component. In case of high throughput, video frames can be sent at higher rates. However, in case of weak throughput, smaller rate may be applied for a quick transfer.

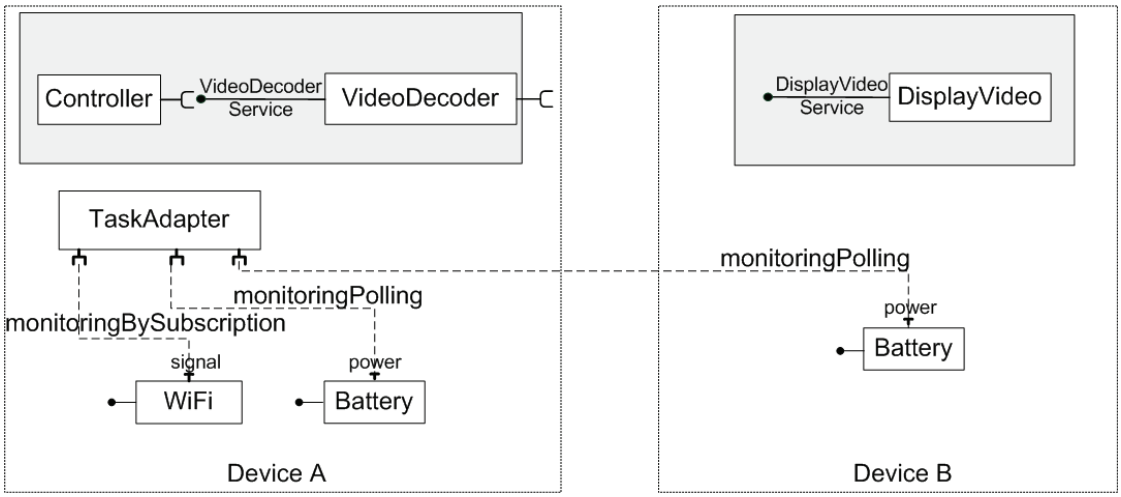


Figure 4.17: Monitoring of required properties of the video player task

The VideoDecoder component may depend also on the changes of battery powers of the communicating devices. If the battery power of the device is low, it uses lower degree of decoding to conserve the battery power. However, if the remaining battery power of each device is above a certain threshold (e.g., 20 percent), higher rate of decoding can be used for a better throughput over the network.

Thus, there is a need to monitor by polling the remaining batteries of the communicating devices and to subscribe to the changes of the network throughput to decide whether the decoded frames rate to apply for the video sent to the DisplayVideo component.

Towards this objective, we need some *adaptive logics* that will make appropriate adaptation decisions based on certain rules for adaptation. This adaptation logic has to be defined by the architect at design time to make the task adaptable. It is encapsulated in an TaskAdapter component. The TaskAdapter component expresses through its required properties its need to monitor local as well as remote properties offered by other components in order to adapt the task.

Figure 4.17 shows the monitoring needs of the video player task. The TaskAdapter component expresses the monitoring needs by subscription mode towards the signal property of a WiFi component in order to adjust the frames sent to the DisplayVideo component. It expresses also its need to observe the remaining batteries of the local and remote devices.

Assume that the WiFi and Battery components do not provide the monitoring services despite the request of the TaskAdapter component. In order to be aware about their changes, we propose to transform them into monitorable components. Figure 4.18 shows the transformation of the WiFi component to a new composite to render its property monitorable by the VideoDecoder component. The created composite exposes in addition to the service of the WiFi component, a PCSubscription service that is provided by Monitoringbysubscription component to allow the TaskAdapter component to subscribe to the changes of the signal property. We used the transformation symbol for a monitoring by subscription mode OnChange, as depicted in Figure 4.13(b), to represent the transformation of the WiFi component.

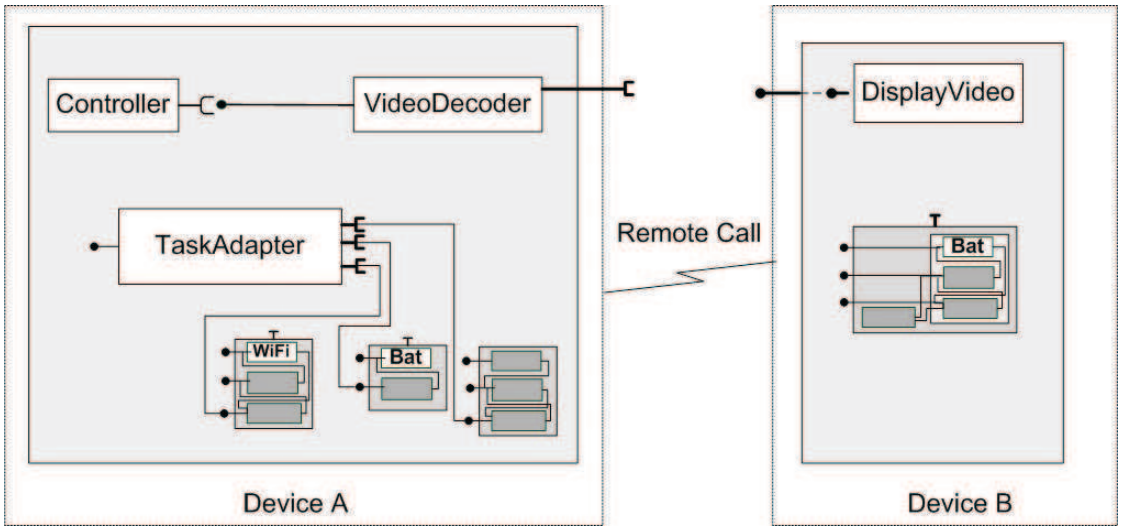


Figure 4.18: Transformation of video player task

Similarly, we have transformed the remote and local Battery components to monitor by polling their powers. For a local monitoring by polling, the Battery component of the device A is encapsulated with a LGenericProxy component in order to observe its changes. We used the transformation symbol for a local monitoring by polling (see Figure 4.9(b)) to represent the transformed Battery component of the device A.

However, to monitor the remote Battery component of the device B, two composites are created for a client and server transformations. In the device A, a ServerProxy component is generated to represent the remote Battery component. This component is encapsulated with a RGenericProxy component to forward its remote calls to the device B following the transformation symbol in Figure 4.16(b). However, in the device B, the created composite consists of an RServer that receives the subscription requests from the RGenericProxy component of the device A, then translates them in order to request the LGenericProxy component about the power property of the Battery component of the device B. The transformation of the Battery component in the device B is modeled following the transformation symbol in Figure 4.15(b).

4.6 Conclusion

In this chapter, we proposed an approach for monitoring of properties of components. The flexibility offered by our approach is that any component that one wants to monitor properties offered by other components, but these latter do not offer these capabilities inherently, can be transformed to offer these functionalities. These aspects are treated independently of the functional code and, hence, do not make the situation more complex for the developers. We proposed an extension to the component concept in order to allow it to express explicitly its dependency to properties offered by other components. Thus, the component is able specify in addition to its provided and required services, and its properties, the required properties that may depend on their changes and needs to monitor them.

This chapter represents a foundation for the next chapter where we will define an adaptation approach for components-based user tasks. In fact, the monitoring approach will be used to capture the events that trigger the adaptation of applications in dynamic environments. In the next chapter, we tackle the adaptation issues given the captured changes in the execution environment.

Chapter 5

User Tasks Adaptation

5.1 Introduction

A Resolution of an abstract user task corresponds to the selection of concrete components provided by various devices of the execution environments. However mapping from abstract to concrete task cannot be done effortlessly; several problems can be detected at init time that prevent the mapping to be achieved successfully, e.g., heterogeneity of network connection interfaces, heterogeneity of interaction protocols, etc. These problems may imply a mismatch between the given abstract description of the task and the concrete level, if the current abstract description of the task could not be realized in the given context.

Moreover, user tasks in pervasive environments are challenged by the dynamicity of their execution environments due to, e.g., users and devices mobility, which make them subject to unforeseen failures and mismatching.

We distinguish two major events categories triggering the adaptation of user tasks in pervasive environments namely, failures and mismatching. The failures may be caused by changes at runtime of user preferences or devices capabilities or due to the mobility

of devices etc. However, the mismatching between abstract and concrete levels can be captured at init time or during the execution of the task to denote the inability of the abstract description to be executed in the given context or in the new context if it changes.

Therefore, there is a crucial need to overcome them and hence to adapt the user tasks to ensure the continuity of their execution.

In literature, two main adaptation techniques are used, which are parametric or compositional mechanisms to adapt applications in pervasive environment (McKinley et al., 2004). Parametrization techniques aim at adjusting internal or global parameters in order to respond to changes in the environment. Compositional adaptation goes far beyond the simple code tuning, and allows exchange of algorithmic and structural parts of the system in order to improve a program's fit to its current environment. It is classified into structural and behavioral adaptations. The behavioral adaptation corresponds to the modification of the functional behavior of an application in response to changes in its execution environment. However, structural adaptation allows the restructuring of the application by adding or removing software entities with respect to its functional logic.

In this chapter, we studied the different contexts triggering the compositional adaptation of user tasks in pervasive environments and we present the corresponding adaptation actions to overcome the failures and mismatches detected at init time or during their execution.

The chapter is organized as following. First, we cite some adaptation approaches in Section 5.2. Then, in Section 5.3, we describe the most prominent adaptation contexts that denote failures or mismatches between abstract and concrete levels. After that, we detail our compositional adaptation approaches that are the reselection of services implementations (in Section 5.4) and restructuring of abstract tasks (in Section 5.5). For each approach, we explain the corresponding adaptation actions to overcome the failures or mismatching illustrated by an example scenario to better explain the principle of each approach. Finally, we conclude the chapter with an overview outlining the most important aspects of our adaptation approaches.

5.2 Compositional Adaptation related Approaches

The problem of adapting component-based applications has been extensively studied in the literature. In this section, we cite approaches that propose solutions to the compositional adaptation issue. We distinguish approaches for reselection of services' implementations and the structural adaptation ones.

5.2.1 Service Reselection Adaptation

QoS-based Reselection

(Li et al., 2010b) proposes an adaptation approach to reconfigure SOA-based application to comply with a new quality of service (QoS) constraint by replacing its individual or multiple component services.

The reconfiguration is driven by the changes of the QoS of SOA-based applications. They distinguish two major causes that can trigger the dynamic reconfiguration. Firstly, if one component service violation happens, thus the service needs to be replaced by an

appropriate candidate in order to ensure an application working by complying with the initial QoS constraint. Secondly, if a user expects to improve the QoS of the application, there is a need to replace components in order to satisfy a given QoS constraint.

In their work, authors focus on the second cause. For this, they propose to adapt the applications in two phases: 1) try to replace each individual component service or 2) replace multiple components services. For the both phases, the adaptation is achieved according to the global significance value. The global significance value is obtained by a weighted quantitative calculation of QoS attributes of each service. In fact, four QoS attributes of Web services are considered as follows: Response Time, Cost, Reliability, Availability.

If the global significance value of one service is biggest, this means that it contribution on the improvement of the QoS of the application is greatest, thus it should be replaced firstly. Therefore, they go through the candidates of the service which have the same or similar functions with it to search for a substitutor. If all candidates cannot comply with the QoS constraint, it will try to replace another service whose global significance value is the highest among the rest of component services and so forth. If all attempts failed in the first phase, the number of the replaced services will be increased gradually to replace multiple component services until the application meets the new QoS constraint required by users.

Hence, using this approach allows to replace components gradually given their global significance values, whenever the user preferences change. However, in their work they do not consider neither the appearance/disappearnce of devices or components nor the changes of devices capabilities which are also prominent effects having an impact on the components replacement. Moreover, identifying the replaced components among a large number of components does not reveal an easy task as they should attempt the an expanded number of replacements in order to identify the suitable one.

Service Process Reconfiguration with End-to-End QoS Constraints

(Zhai et al., 2009) presents an approach for repairing failed services by replacing them with new ones and ensuring that they meet the user specified end-to-end QoS constraints. The reconfiguration is triggered by services failures to deliver the QoS as requested by users because of, for example, network delay, host overload, unexpected inputs, etc.

Their proposal consists of a reconfiguration algorithm that is designed to produce reconfiguration regions that include one or more failed services. When one or more services in a service process fail at runtime, they try to replace only those failed services in that a way, the resulting service process must still comply with the original end-to-end QoS constraints.

The algorithm first identifies the regions that are affected by faulty services. The motivation from the reconfiguration region is to reduce the number of services that need to be replaced by the service recomposition algorithm. In this algorithm, the region identification procedure increases the size of affected region gradually. The range bound starts from zero, which means the algorithm will try to replace only the failed services first. For a failed service, an output reconfiguration region extends from it to all nodes that are connected to it. If it fails to replace the services successfully, the range bound will be increased, therefore, more services will be added to the region to be reconfigured.

The algorithm will then try to recompose these regions. This recomposition is a multi-QoS attribute service composition problem with end-to-end QoS constraints. After the initial recomposition attempt, if there are still some regions that cannot be reconfigured, the algorithm will increase the range of the affected regions, which means more nearby services will be added into the region. The algorithm continues until all regions have a satisfactory subprocess composition.

Thus, the adaptation is ensured by identifying services in reconfiguration regions and attempting to replace them one by one until the end-to-end QoS constraints are satisfied. However, this task does not seem to be trivial for a region with an expanded number of services.

Reconfigurations Plans

MADAM (Mobility and ADaption enAbling Middleware) (Floch et al., 2006) proposes to reconfigure applications by using plans. A plan describes an alternative application configuration. It mainly consists of a structure that reflects the type of the component implementation and the QoS properties associated to the services it provides. To select the suitable configurations, they are ranked by evaluating their utility with regards to the application objectives and the user preferences in order to select a configuration with a highest utility. However, these configuration plans are predefined by a developer at design time, which limits the possible adaptations to these predefined plans. Moreover, the utility function is limited to the application and hence can not be used for other applications.

The PCOM system (Becker et al., 2004a) also uses reconfiguration plan algorithms to support a PCOM component reselection whenever this later is no longer available. With respect to the application model defined by PCOM, which is modeled as a tree, the reconfiguration consists of escalating to the parent component. The escalation continues until the conflict is resolved by reselecting components. Thus, the replacement of a subtree starts from the predecessor of the component and may include its successors if it is necessary even if they are not concerned by the changes.

5.2.2 Structural Adaptation

Adaptation of Components Interfaces' Mismatches

In (Becker et al., 2004b), authors propose an adaptation approach which is built upon a classification of components interfaces' mismatches. They introduced a taxonomy to enumerate different types of component mismatches which will be taken into consideration for the adaptation.

As drawn in Figure 5.1, they distinguish the Signatures mismatches, which is caused by different signatures of components interfaces (methods' names, parameters' types, etc.). The mismatches between Assertions (i.e. pre- and postconditions for the methods) of interfaces are captured when comparing provided and required interfaces. Protocols mismatches are related to the messages of the interacted components for example ordering of messages, absence of messages etc. The Quality Attributes mismatches are captured if the provided and required interfaces do not make the same QoS assumptions about the authentication, access, and integrity of messages, etc. The Concepts mismatches are

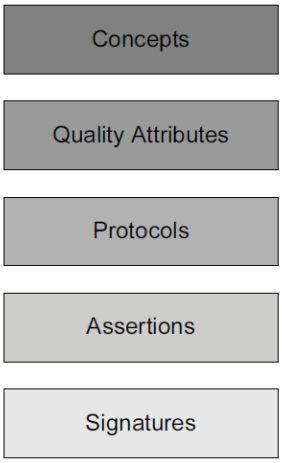


Figure 5.1: Classification of interfaces mismatches (Becker et al., 2004b)

detected if for example corresponding interface elements of a provided and required interface, are identical with respect to their definition, but have been used with different terms (e.g. customer and buyer).

To cover these mismatches, they identify a number of patterns to be used for eliminating the interfaces' mismatches. They classify the adaptation patterns into two categories: functional adaptation pattern to bridge the functional component incompatibilities and extra functional adaptation pattern to increase a single or several QoS of the component being adapted. For example, the Adapter, Decorator and Bridge patterns (Gamma et al., 1995) are used to bridge the functional component incompatibilities and to ensure the delegation functionality. However, the extra-functional adaptation patterns consists of an increasing a single or several quality of attributes of the components being adapted like security, authentication, encryption etc.

The major drawback of these adaptation patterns that are limited to the mismatches between components' interfaces and they do not take into account the network and hardware heterogeneities of devices.

AO-ADL

(Pinto and Fuentes, 2007)(Fuentes et al., 2007) present an Aspect Oriented-Architecture Description Language (AO-ADL) to model applications. The main architectural elements of AO-ADL are components and connectors. AO-ADL considers that components having either crosscutting (named aspectual component) or non-crosscutting behavior (named base component) exhibiting a symmetric decomposition model. The aspectual components are used to adapt an AO-ADL application by inserting crosscutting behaviour with respect to the functional behaviour of the application.

Any component (i.e., base or aspectual component) is described in terms of offered services, required services and parameters. AO-ADL components may play an aspectual or non-aspectual role depending on the particular interactions in which they participates. This role serves as an identifier for the component or aspect and is used for coupling between components.

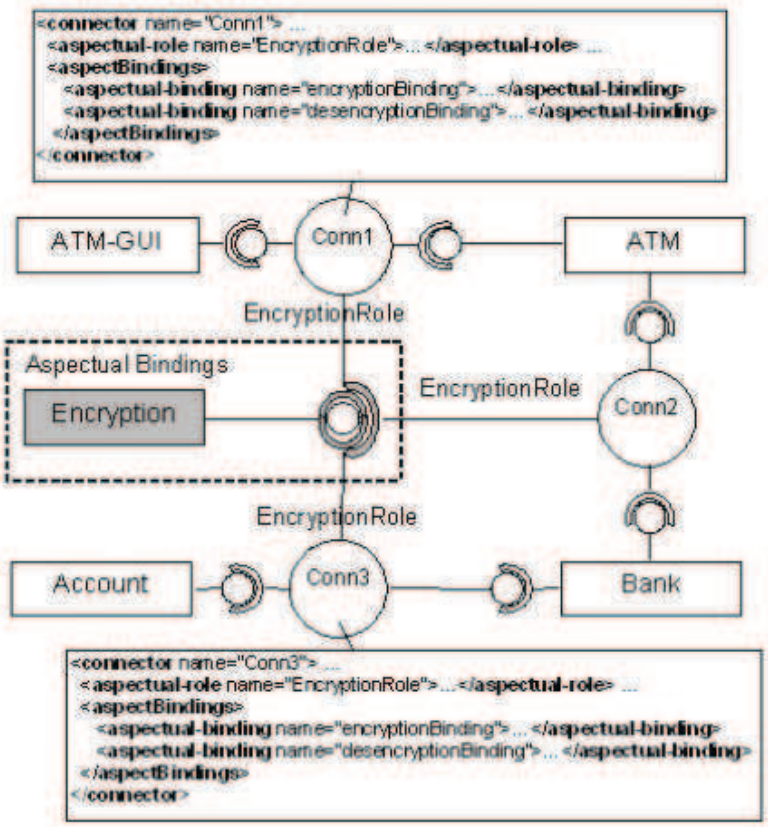


Figure 5.2: Example of an aspectual composition

The binding between components is specified in AO-ADL connectors. AO-ADL connectors specify how aspectual components are weaved with base components during components' communication. They distinguish two different binding to differentiate between aspectual components and base components. A component binding describes the connection between the base components, while the aspectual binding specifies the name of an aspectual component and a pointcut expression. These connectors are described following templates that are specific to them.

Figure 5.2 shows an aspectual component which is used to ensure the security of transactions in Automated Teller Machines (ATMs) system for a bank. The Encryption component, which is included as an aspectual component, affects the interactions between the ATM-GUI and the ATM components, between the ATM and the Bank components and between the Bank and the Account components. These connections are specified by three different connectors. Consequently, an aspectual-role clause specifying the encryption aspectual behavior and an aspectual-binding clause specifying how encryption is bound to the interaction among base components have to be replicated in the three connectors.

However, the instantiation of the connector template is done at design time, which limits the possibility to extend applications with new aspectual components. Moreover,

they consider a connector template per aspect. However, it is not possible to define as many connector templates as possible aspects.

Mismatch patterns

(Kongdenfha et al., 2009) (Benatallah et al., 2005) have investigated matching of Web service interfaces by providing a classification of common mismatches between service interfaces and business protocols, and introducing mismatch patterns. These patterns are used to formalize the recurring problems related to the interaction between services.

Each mismatch pattern has a name and a mismatch type that provides a description of the mismatch that is captured. A mismatch pattern includes a template of adaptation logic that resolves the detected mismatch.

Their proposal is to use aspect-oriented approach to specify the mismatches patterns. In this case, each mismatch pattern consists of advice templates which are implemented using XQuery templates. To instantiate adaptation advices, developers need to provide transformations functions, i.e., XQuery functions, to the advice templates. While instantiating each adaptation advice, developers also specify the process execution point (joinpoints) where the advice needs to be executed. Joinpoints are specified in terms of queries on the process code. Both the joinpoint queries and their corresponding advice for each of the mismatch are described in a single file called the Aspect Definition Document (ADD).

However, their proposed mismatch patterns are limited to the interfaces and protocols mismatches. Moreover, the specification of adapters supplies some pseudo code predefined by the developers. Thus, it is not possible to integrate them dynamically at runtime without the assistance of the developers.

Mediating Connectors

In (Spalazzese and Inverardi, 2010), authors propose an adaptation approach to allow the interoperability between components supporting different interaction protocols at runtime. Their approach is based on mediating connectors that are used to dynamically cope with components' behavioral diversity.

The approach first decomposes the components' behavior into elementary behaviors representing elementary intents of the components. Then, there is a need to check the possibility for the two protocols to communicate. If the two protocols are not complementary, the framework should identify the mismatches in order to find out the suitable mediating connector to use.

Towards these mismatches, they propose a list of basic mediator patterns given a mediator connector template. These patterns are: (1) Message Consumer Pattern, (2) Message Producer Pattern, (3) Message Translator Pattern, (4) Messages Ordering Pattern, (5) Message Splitting Pattern, (6) Messages Merger Pattern.

Based on the identified mismatches and their relative solutions, they intend to define a composition strategy to build a mediating connector's behavior starting from the elementary mediating behaviors.

This preliminary work introduces Connect, a software framework which aims to resolve the interpretability challenge for dynamic application domains. The mediators will be used to resolve a specific components' mismatches, which are related to protocols'

heterogeneity. The specification and realization of mediators remain a challenge.

In (Li et al., 2010a), authors propose too to use the mediation patterns to overcome behavioral mismatches for web services interaction. They categorize the mediation on two levels, signature and protocol mediations. Signature mediation focus on message types, methods names, etc. In comparison, protocol mediation focus on resolving mismatches occurring at the message exchanging sequences. They have proposed six basic mismatch patterns and pointed out that all possible protocol mismatches can be composed by these basic patterns.

The basic mediators are namely, (1)Storer pattern, (2) Constructor pattern, (3) Merger pattern, (4) Splitter pattern (5) Storing Controller pattern and (6) Constructing Controller pattern.

The purpose is to introduce mediators when composing services together in order to make both of the requester and provider interfaces fit each other smoothly. Each mediator is based on transformation rules to deal with these interfaces. Therefore, a mediator pattern consists of input interfaces that are used to receive messages sent from a service requestor, output interfaces that are used to send messages to a provider and a rule engine to determine which transformation rule defined in the rule base should be activated. However, these proposed mediators connectors are limited to the protocol or signature level.

5.3 Adaptation Context

We are interested in the compositional adaptation techniques. In this section, we put the light on the most important contexts that may trigger either the reselection of services' implementations or the structural adaptation of an abstract description of a user task.

A generalized notion of context has been proposed in (Abowd et al., 1999) as *any information that can be used to characterize the situation of an entity (person, location, object, etc.)*. We consider adaptation context as any piece of information that may trigger the adaptation of the task.

For a reselection adaptation, a context may represent an event corresponding to the disappearance of a device, which may include components used for the execution of a user task. In this case, the disappearance of the device leads to the failure of the task execution as the used components are disappeared. Similarly, the disappearance of components because of their undeployement implies the failure of the task if they are selected for the execution of a user task.

To trigger the reselection of services, an adaptation context may imply the appearance of a new device or component that may be interesting for the execution of the user task. In this case, there is a need to adapt the user task in order to use the new components. Moreover, the changes of user preferences or devices capabilities may have an impact on the execution of the user task.

We are also interested in contexts that represent mismatches between an abstract description of a user task and the concrete level to trigger a structural adaptation. These mismatches imply that the current abstract description could not be realized in the given context, or in the new context, if it has changed.

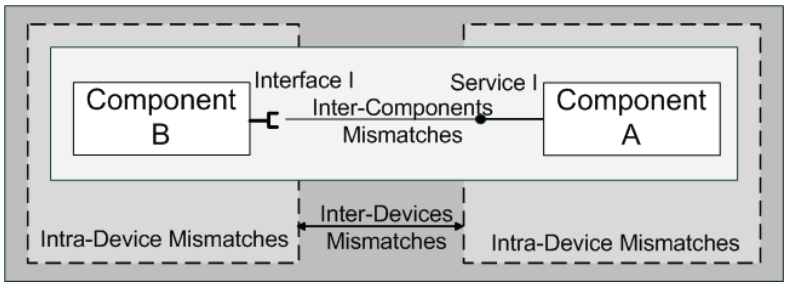


Figure 5.3: Categorization of mismatches levels

We have classified the mismatches level, where they occur, into three categories: inter-components, intra-device and inter-devices mismatches as shown in Figure 5.3.

At the inter-components level, we consider the mismatches that may arise at init time due to the non-satisfaction of the extra-functional requirements of the components. These latter are system requirements which are not of a functional nature, but contribute decisively to the applicability of the system (Galster and Bucherer, 2008) like security, reliability, etc. Thus, if they are not ensured at the concrete level, this may prevent the abstract task to be well mapped. At this level, the mismatches may be also related to the heterogeneity of signatures, protocols or semantic of services (Becker et al., 2004b). In the present work, we do not focus on these mismatches.

It is also possible to detect mismatches at intra-device level. These mismatches denote that the desired characteristics of the devices are changed at runtime like using reduced capacities as a congested memory, a slower CPU, etc. Thus, there is a need to adapt the user task at runtime to consider these changes.

In case of a distributed environment, there is also a need to consider the mismatches occurring at inter-devices level. These mismatches may be caused by the heterogeneity of network characteristics of communicating devices for example, using heterogeneous interfaces of connection. Thus, they have an impact in the communication between devices.

In the following, we present the different adaptation actions to overcome the captured failures and mismatches and hence ensure a continuous execution of the task.

5.4 Partial Reselection Adaptation

In this section, we propose a dynamic compositional adaptation approach by replacing services implementations at runtime for a continuous execution of a user task. The reselection of services is triggered by the failures of task execution due to the disappearance/appearance of devices or components, the changes of user preferences or devices capabilities. The adaptation consists of a partial reselection of services implementations since there may be cases in which components are not affected by the contexts' changes. We describe in the following the adaptation actions related to the most prominent events.

5.4.1 Adaptation actions

For each adaptation context, we defined the corresponding actions (Ben Lahmar et al., 2012) as following:

Disappearance of a used device If a device disappears from an execution environment, this may imply the disappearance of components selected for the execution of a user task. Therefore, there is a need to check whether the disappeared device was selected for the execution of the user task or not. We have considered previously that the resolution of an abstract task is achieved in two steps; 1) selection of devices for each service of the user task and 2) selection of concrete components therein (see chapter 3). As a result of the device selection phase is a device table that ranks devices, for each service of the task, from high to low values.

According to this table, it is possible to determine if the disappeared device has the highest value or not. If it is, we propose the selection of a subsequent device in the devices table and then select its components to fulfill the replacement of services implementations. However, if the disappeared device has not the highest value in the devices list for a task's service, there is no need to trigger the adaptation of the user task and we require only removing it from the table (see Algorithm 5.1).

Algorithm 5.1 DeviceDisappearanceAdaptation(Task t, DevicesTable dt, Device d)

```
1: for each service s_i ∈ t do
2:   if dt(s_i)[1].d==d then
3:     if dt(s_i)[2] ≠ Null then
4:       Select dt(s_i)[2].d for s_i
5:       ComponentSelection(s_i, dt(s_i)[2].d)
6:     end if
7:     remove d and its corresponding value from the dt(s_i)
8:   end if
9: end for
```

Appearance of a new device The appearance of a new device may trigger the adaptation of a user task, if some of the task's services fit in and its capabilities ensure their requirements. Indeed, a new device appearing in the execution environment may be interesting for the execution of some services of the task, if it has a highest device value. The fact that it has a higher value than the existing devices, means that it responds better to the services requirements and to the user preferences.

Therefore, we require to calculate its value for each service of the task fitting in, to determine whether it has a higher value than the value of the currently used device. If it is, the new device will replace the already selected device to map the services with its provided components. Consequently, there is a need to select the convenient components to achieve the services reselection (see Algorithm 5.2).

Disappearance of a component Another important event that may trigger the adaptation of a user task is the disappearance of a component from a selected device. The disappeared component may not be selected for the execution of a service of the user task. Therefore, there is a need to check in the components table associated to that device whether the disappeared component was selected for the execution of a task's service. If it is not, there is no need to reselect a new component for the service's execution. Otherwise, we propose to select the subsequent component in components table associated

Algorithm 5.2 DeviceAppearanceAdaptation(Task t, DevicesTable dt, Device d)

```
1: for each service s_i ∈ t do
2:   if fit(s_i, d) and fit(colocation(s_i), d) then
3:     calculate DV the device value of d for s_i
4:     if DV ≥ dt(s_i)[1].DV then
5:       Select device d
6:       ComponentSelection(s_i, d)
7:     end if
8:     insert the couple < d, DV > in the dt(s_i)
9:   end if
10: end for
```

Algorithm 5.3 ComponentDisappearanceAdaptation(Task t, ComponentsTable ct, DevicesTable dt, Device d, Component c)

```
1: for each service s_i ∈ t do
2:   if d was selected for s_i then
3:     if c == ct(s_i)[1].c then
4:       if ct(s_i)[2] ≠ Null then
5:         Select ct(s_i)[2] for the execution of s_i
6:       else
7:         select dt(s_i)[2].d for s_i
8:         ComponentSelection(s_i, dt(s_i)[2].d)
9:       end if
10:      remove c and its corresponding value from ct(s_i)
11:    end if
12:  end if
13: end for
```

with the service, if the device provides more than one component functionally similar to the disappeared one (see Algorithm 5.3). But, if the device has no more components matching with the disappeared component, there is a need to select the subsequent device in the devices table and consequently select a component matching with the disappeared one therein.

Appearance of a new component in a selected device During the execution of a user task, a new component may be deployed on a device selected for the execution of a user task. The new component can be interesting for the execution of the user task if its description matches with a service of the user task. If it is, we require calculating its value in order to compare this latter with the value of the already selected component. If the new component has the highest value, it will be selected for the execution of the service of the task. Otherwise, the currently used component is kept and the new one is added to the ranked components table associated with the service for a future use (see Algorithm 5.4).

Algorithm 5.4 ComponentAppearanceAdaptation(Task t, ComponentsTable ct, DevicesTable dt, Device d, Component c)

```
1: for each service s_i ∈ t do
2:   if d was selected for s_i then
3:     if s_i matches c then
4:       CV=calculateCV(c, DC)
5:       if CV ≥ ct(s_i)[1].CV then
6:         select c for the execution of s_i in d
7:       end if
8:       insert the couple c and its corresponding value in ct(s_i)
9:     end if
10:  end if
11: end for
```

Algorithm 5.5 ChangesOfDevicesValuesAdaptation(Task t, DevicesTable dt, Device d, Property p)

```
1: p is a property representing a changed capability for a device d
2: for each service s_i ∈ t do
3:   if p is related to the Requirement(s_i) then
4:     calculate newDV the new device value of d for
5:     if d==DT(s_i)[1].d then
6:       if newDV ≤ DT(s_i)[2].DV then
7:         Select DT(s_i)[2].d for s_i
8:         ComponentSelection(s_i, DT(s_i)[2].d)
9:       end if
10:    else
11:      if newDV ≥ DT(s_i)[1].DV then
12:        Select d for s_i
13:        ComponentSelection(s_i, d)
14:      end if
15:    end if
16:    rank DT(s_i)
17:  end if
18: end for
```

Changes of a device capabilities or user preferences The changes of the user preferences or devices capabilities may trigger the adaptation of a user task if they are related to its services requirements. Indeed, for each service of the task, the devices values are calculated according to the user preferences and devices capabilities related to its requirements. Thus, changing the user preferences or devices capabilities or the both imply the changes of devices values from low to high values and vice versa.

Thus, if a device capability or a user preference related to a service requirement changes, there is a need to recalculate the new device value. If a device is already in use for the execution of some services of the user task and its value changes from high to low, we propose to select the subsequent device in the devices table associated to the considered services. However, if a device is not used beforehand and its value becomes higher than the value of the used device, it will be selected to assign the concerned services to its components (see Algorithm 5.5).

In summary, our adaptation approach allows the partial reselection of concrete components for a continuous execution of users tasks in pervasive environments. The main advantage of this approach is that the adaptation is limited to the services that are affected by the changes of the execution environment, thus, there is no need to adapt the whole task.

5.4.2 Example Scenario

Referring back to the video player task in the introductory Chapter 1, we consider that the task is represented by an assembly of three services: a VideoDecoder, a DisplayVideo and a Controller services.

Table 5.1: Device capabilities and user preferences

Capabilities	SP	FS	User
Resource.Software.VideoPlayer=VLC	1	0	0.1
Resource.Hardware.Output.VideoCapable.Screen.Width	320	1920	1920
Resource.Hardware.Output.VideoCapable.Screen.Height	240	1200	1200
Resource.Hardware.Output.SoundCapable.InternalSpeaker	1	1	0.2
Resource.Hardware.Output.SoundCapable.ExternalSpeaker	1	1	1.0
Resource.Hardware.Input.Keyboard	1	0	0.1
Resource.Hardware.Input.TouchScreen	1	0	0.3
Resource.Hardware.Memory.MainMemory	1	1	0.0
Resource.Hardware.Memory.Disk	1	0	0.0

For the task execution, the services should be resolved in concrete components available in the execution environment. Consider that this later consists of a Smartphone (SP) and Flat-screen (FS) devices having the following capabilities in Table 5.1.

Moreover, services may express their requirements towards the devices capabilities. For example in table 5.2, the DisplayVideo service requires an output VideoCapable

and SoundCapable capabilities to achieve its execution, whereas, the Controller service requires an input capability to command the video player task.

Table 5.2: Requirements of the video player services

Service	Service Requirements
Controller	Resource.Hardware.Input
DisplayVideo	Resource.Hardware.Output.VideoCapable
	Resource.Hardware.Output.SoundCapable

Using the resolution approach in Chapter 3, the task's services will be executed as the following: the Controller service will be executed in SP device, while the DisplayVideo and VideoDecoder services will be executed in FS device given the devices values in table 5.3.

Table 5.3: Devices values for Controller and DisplayVideo services

Service	SP	FS	Selected device
Controller	1.4	-0,4	SP
DisplayVideo & VideoDecoder	5.16	5.5	FS

Assume that during the execution of the user task, the FS device capability is changed in that a way it does not provide an external speaker capability. This change may have an impact of the device's value since the DisplayVideo service specifies a sound capable requirement. Therefore, there is a need to recalculate the device value for the DisplayVideo service as it depends on the changes of this capability.

Table 5.4: Devices values for VideoDecoder and DisplayVideo services after changes of the FS's capability

Service	SP	FS	Selected device
DisplayVideo & VideoDecoder	5.16	0.5	SP

Table 5.4 shows the device value of SP and the new the device value of FS As it can be seen, the changes of the FS capability has induced the change of its value from high (5.5) to low value (0.5). The new selected device is the SP as it has the highest device value (5.16).

In another context, assume now that during the execution of the video player task, a laptop device appears. Thus, there is a need to determine whether it is useful for the execution of some of services of the video player task. Table 5.5 describes the capabilities

Table 5.5: Capabilities of a laptop device

Capabilities	LP
Resource.Software.VideoPlayer=VLC	1
Resource.Hardware.Output.VideoCapable.Screen.Width	1280
Resource.Hardware.Output.VideoCapable.Screen.Height	800
Resource.Hardware.Output.SoundCapable.InternalSpeaker	1
Resource.Hardware.Output.SoundCapable.ExternalSpeaker	1
Resource.Hardware.Input.Keyboard	1
Resource.Hardware.Input.TouchScreen	0
Resource.Hardware.Memory.MainMemory	1
Resource.Hardware.Memory.Disk	1

of the new device (i.e., LP). As it can be seen, the capabilities of the laptop device include the services requirements of the user task's services.

Table 5.6: The laptop value for the video player services

Service	LP
Controller	.2
DisplayVideo & VideoDecoder	5.41

To determine the device usefulness, there is a need to calculate its value for each service of the user task. Table 5.6 shows the values of the LP device. As it can be seen, the LP device has a higher value (5.41) than the FS device value (0.5) and SP one (5.16), thus, it will be selected for the execution of the DisplayVideo and VideoDecoder services. However, there is no need to reselect the implementation of Controller service as the value of LP is less than the value of the SP device.

5.5 Structural Adaptation

In some adaptation contexts, reselecting new components is not sufficient to recover the failures. This is the case of the situations triggered by mismatches between an abstract description of a user task and a concrete level, implying that the task could not be realized in the given context, or in the new context, if it has changes (as detailed in Section 5.3).

In this section, we propose a dynamic compositional adaptation approach that consists of restructuring the abstract description of user tasks with respect to their functional behaviour. The adaptation can be done at runtime to ensure a continuous execution, or at init time to fulfil their mapping. We describe in the following the principle of our structural adaptation approach.

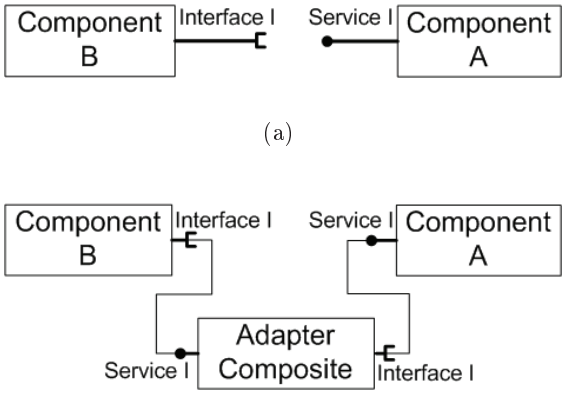


Figure 5.4: Transforming abstract task using Adapter Composite

5.5.1 Principle of the Structural Adaptation Approach

To overcome a captured mismatch between an abstract description of a user task and the concrete level, there is a need to adapt the abstract description to ensure its mapping and its execution.

In (Ben Lahmar et al., 2010a), we proposed a dynamic structural adaptation approach for abstract user tasks, which consists of transforming a task to another one that allows its mapping and execution. The transformation is ensured by injecting some adapters exhibiting extra-functional behaviours into the user task without modifying its functional behaviour.

For example, as shown in Figure 5.4, an Adapter Composite is injected to adapt the communication between components A and B. The Adapter Composite requires the service I of the component A and exposes a service implementing the interface I. This provided service will be used by the component B, since it corresponds to its required service.

5.5.2 Adaptation Patterns

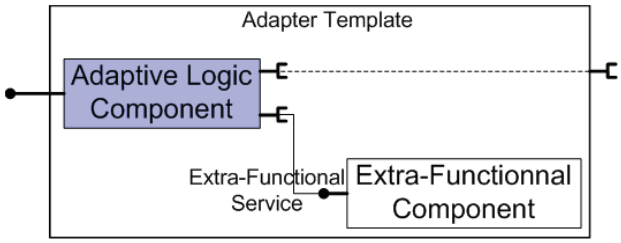


Figure 5.5: Adapter Template structure

As the basis for our approach, we have proposed to use adaptation patterns as adapter composites, which provide solutions to overcome mismatches between an abstract descrip-

tion of user task and a concrete level and can be used in different and recurrent contexts (Ben Lahmar et al., 2010a).

From a design pattern standpoint, an adaptation pattern corresponds to a proxy as defined in (Gamma et al., 1995) since it represents a component and provides the same service as required by the caller components.

We have defined an Adapter Template to be used for the description of adaptation patterns. Figure 5.5 shows the description of the adapter template, which consists of an "adaptive logic" and an "extra-functional" components.

The extra-functional component provides services allowing, e.g., encryption, compression, etc. It has an abstract description to be mapped to concrete level (see chapter 3). Thus, the extra-functional component is resolved dynamically without the assistance of designer.

The adaptive logic component is considered as the main element of the adapter template. Thus, each adaptation pattern should contain at least the adaptive logic component, if it does not require any extra-functional service to overcome the mismatch. The adaptive logic component encapsulates the adaptation logic and uses the extra-functional service. This component has a generated implementation as it depends to the interfaces of communicated components. As shown in figure 5.5, the adaptive logic component relies on the offered service of the extra-functional component to ensure the adaptation.

Using this specific structure of the adapter template has an advantage, on one hand, to separate the adaptive logic from the functional logic of a user task. Thus, it is possible to modify the adaptive logic with respect to the components' descriptions and to the adaptation contexts. On the other hand, it allows providing an abstract description for the adaptation actions, corresponding to extra-functional components, to be mapped to the concrete level. Thereby, the separation between adaptive logic and extra-functional components facilitates the generation of the adaptive logic.

In the following, we present a set of adaptation patterns that are defined using the Adapter Template (Ben Lahmar et al., 2011a). For each pattern, we give its description, the context in which it will be used, and where it will be used to overcome a mismatch.

The list of the adaptation patterns is not exhaustive. However, it is possible to define other adaptation patterns following our adapter template.

Encryption and Decryption Adaptation Patterns

Description we propose an encryption pattern that intercepts the interaction between components to encrypt messages transiting over a network in order to prevent the disclosure of information to unauthorized components. To use the original message, there is a need to restore it from the encrypted one. For this purpose, we also define a decryption pattern that decrypts the received messages before using it by the target component.

The encryption and decryption patterns consist of an adaptive logic component and an extra-functional one as shown in the figure 5.6. Each extra-functional component exposes a key property and provides a service ensuring the encryption or decryption of a message.

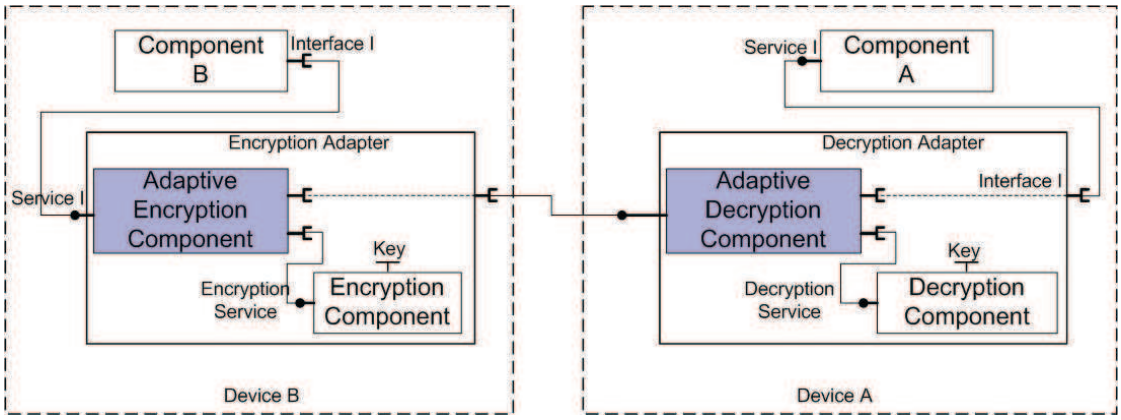


Figure 5.6: Encryption and Decryption adaptation patterns

In case of a symmetric encryption and decryption, the transmitter and the receiver sides use the same key to exchange messages. However, if the extra-functional components implement an asymmetric algorithm, they use two different keys.

Adaptation context The encryption and decryption patterns will be injected at init time to ensure the security of the transferred messages, which may be sensitive to disclose as with credit card numbers and passwords.

The security requirement can be expressed explicitly through the extra-functional requirements of services. At init time, if the concrete components do not ensure this requirement as specified in the abstract level, there is a need to adapt the task in order to achieve its mapping.

Where to use The encryption pattern is used by the transmitter device to send encrypted messages, while the decryption pattern is used by the receiver side to restore the messages. For example, in Figure 5.6, an encryption pattern is used by the device B in order to encrypt the messages sent from the component B over the network. However, a decryption pattern was handled in the device A to restore the original message before using it by the component A.

Authentication and Integrity Patterns

Description The authentication pattern is used to sign the transferred message between two components in order to ensure that a message has not been tampered with. The extra-functional component of the pattern generates a signature digest and then encrypt it with a private key in order to add it at the end of the message before sending this later by the adaptive logic component over the network.

In the receiver side, there is a need to validate the authentication of the message's signature to authorize component to invoke the requested service. Therefore, we propose an integrity pattern that will prove the validity of the transmitted message before transferring it to the intended component. Its extra-functional component, as shown in Figure 5.7, returns a boolean result implying the validity of signed messages.

The verification is done by decrypting the digest using the public key of the sender. Then, the extra-functional component of the integrity pattern compares the decrypted digest with a calculated one. If the calculated digest matches the decrypted one from

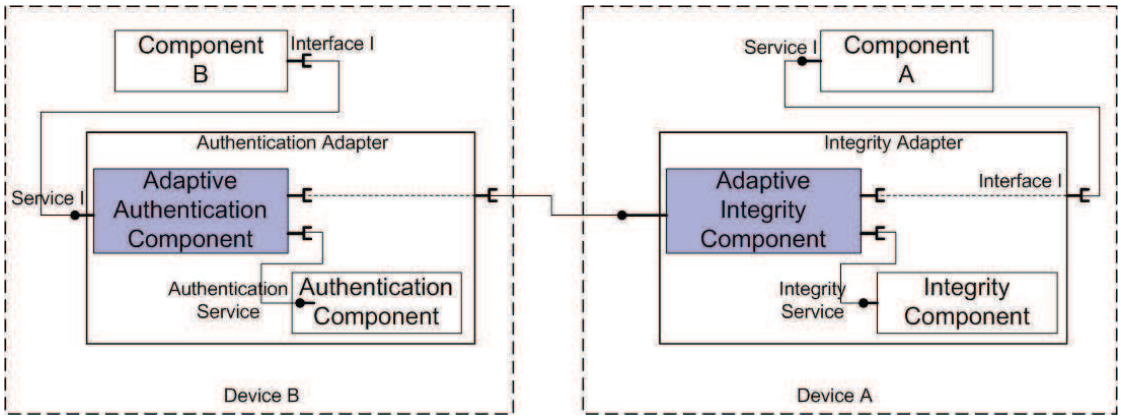


Figure 5.7: Authentication and Integrity adaptation patterns

the received signature, then the integrity is ensured. Otherwise, the message is tampered with during its transfer.

Adaptation context An abstract component may specify at init time through its extra-functional requirements the need to authenticate the transferred messages. If the communicating concrete components do not ensure the authentication and the integrity of messages, this implies a mismatch between the abstract description and the concrete level. Thus, it triggers the adaptation of the user task.

Where to use The authentication pattern is used by the transmitter side to send signed messages. In the receiver side, the integrity pattern will be used to check if the message is kept intact during its transfer over the network. Figure 5.7 shows an authentication pattern that is used by device B to send signed message from the component B to the component A. However, an integrity pattern is used by the device A, to validate the received message from the component B.

Splitting and Merging Patterns

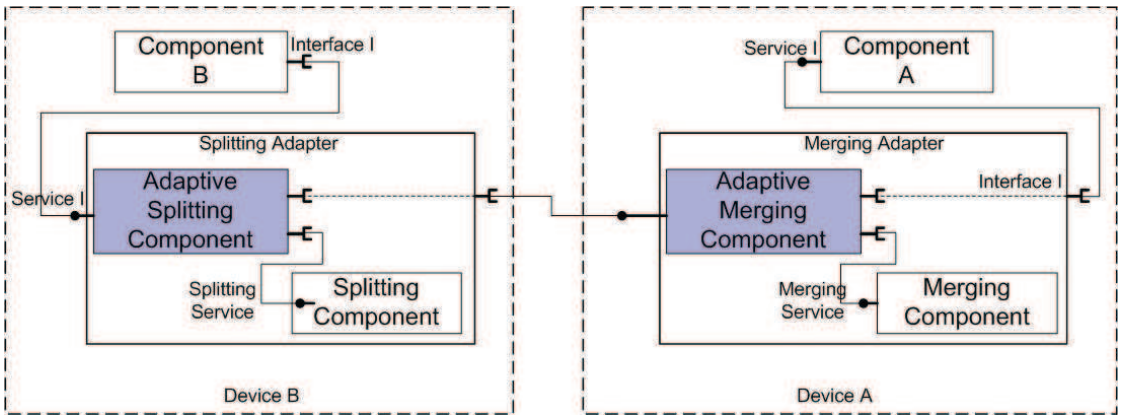


Figure 5.8: Splitting and Merging adaptation patterns

Description The splitting pattern is used to split a message into chunks. The pattern consists of an adaptive logic component and an extra-functional one that returns a list of chunks to send over the network as shown in Figure 5.8.

To respond to the component's request, there is a need to merge the chunks beforehand. Therefore, we propose to compose the splitter pattern with a merger one to form the message from the received chunks. Hence, the extra-functional component of the merger pattern will construct messages from the received chunks, which are forwarded by its adaptive logic component to the intended component.

Adaptation context The splitting and merging patterns may be used to overcome a problem related to a lower network signal captured during the task execution. The change of context has certainly an impact on the message delay in case of the transfer of bigger files or messages. In this case, the message is split into chunks for a quick transfer over a network and hence to reduce the message delay. And a merging pattern will be used by the receiver side to construct it.

Where to use The splitting pattern will be used by the transmitter device, while the merging pattern will be used by the receiver device to fulfill the interaction between devices. Thus, the component B in figure 5.8 is able to send a message or a file into chunks to the component A for a quick transfer over a lower network signal.

Compression and Decompression Patterns

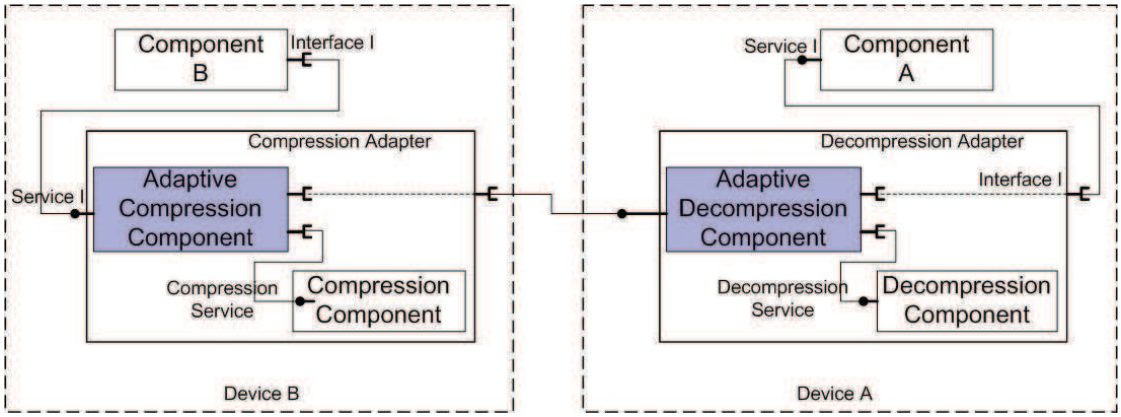


Figure 5.9: Compression and Decompression adaptation patterns

Description The compression pattern is introduced between two components communicating with each other over a network in order to send compressed messages. However, in the receiver device, there is a need to decompress the message in order to be used by the target component. Therefore, we propose to compose the compression pattern with a decompression one to decompress the data before using it by the target component.

Each adapter consists of an adaptive logic component that relies on a non-functional component to compress or decompress the message, as shown in Figure 5.9.

Adaptation Context The compression and decompression patterns are introduced in response to a trigger generated by fluctuation in network QoS. For example, two components exchanging data may be adapted to use the compressor if the network latency or throughput falls below a certain threshold. By using the compression between the

components, all the data will be compressed before transmission over network, allowing efficient transfer of data. Thus, it is useful to overcome a mismatch induced by a network problem.

Where to use We propose to use the compression pattern by the sender device in order to send compressed messages over network. However, the decompression pattern should be handled in the receiver side to decompress the messages before using them as shown in figure 5.9.

Proxy Pattern

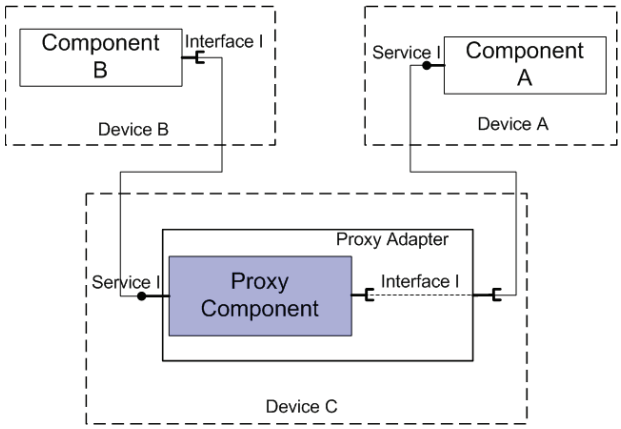


Figure 5.10: Proxy adaptation pattern

Description The proxy pattern allows components to access to services offered by others components. Figure 5.10 shows a description of the proxy pattern following the adapter template. As it can be seen, the proxy pattern represents a specific case of the adapter template as it contains only an adaptive logic component that forwards the call of the service I to the component A.

Adaptation Context The proxy pattern is useful to overcome the network factor related to the heterogeneity of network interfaces. For example, if two devices were selected to map an abstract task and they support two different connection interfaces, e.g. Bluetooth and wifi, thus, the mapping will fail. Therefore, we propose to introduce the proxy pattern, in a device supporting the both network interfaces (i.e., Bluetooth and wifi), to act as an intermediate between the communicating components.

Where to use To intermediate the communication between devices, we propose to generate the proxy in a third device. Thus, the components A and B, as shown in figure 5.10, can communicate together via the proxy generated in a device C.

Caching Pattern

Description the caching pattern enables a user task to cache messages in rapid memory. Figure 5.11 shows the main features of the caching pattern. It consists of an adaptive logic component that will first check the cache to see for example if the response of the component request can be found there. Failing to find the response in the cache, the adaptive caching component will forward the call to the target component. Once it

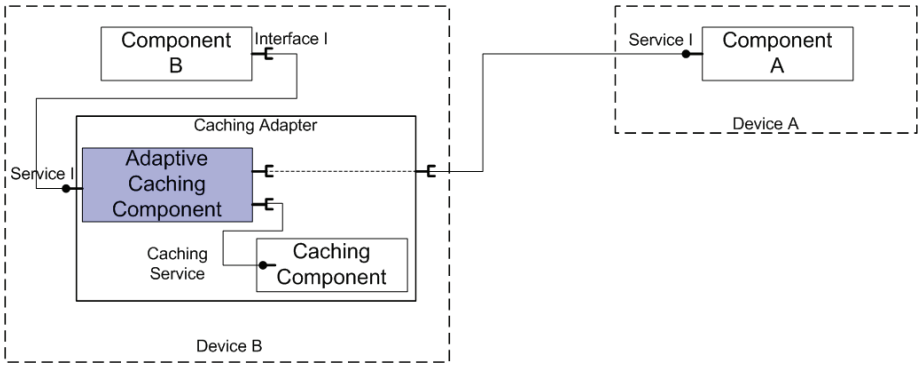


Figure 5.11: Caching adaptation pattern

receives the response, it will forward it to the caller component after storing it in the cache. The caching service provides mainly methods for retrieving, updating and setting message in the cache.

Adaptation Context The caching pattern can be used during the execution of a user task to avoid the congestion of a network by storing the responses to the services' requests. Therefore, the system should monitor the latency of the used network to identify if there is not a jitter. Otherwise, a caching pattern will be injected during the execution of the task to avoid the congestion for a further uses.

Moreover, some component may express through their non-functional requirements the need to have a decreased response time, for example the response time of service I is less than 50 ms. If the concrete component does not consider this requirement, there is a need to inject a caching pattern at init time in order to try to decrease the response time during the execution of a user task.

Where to use The caching pattern will be used either by the sender side where the message will be stored. For example, in Figure 5.11, the pattern is used to decrease the response time to the requests of the component B by caching the call to service I in a cache of the device B.

Retransmission pattern

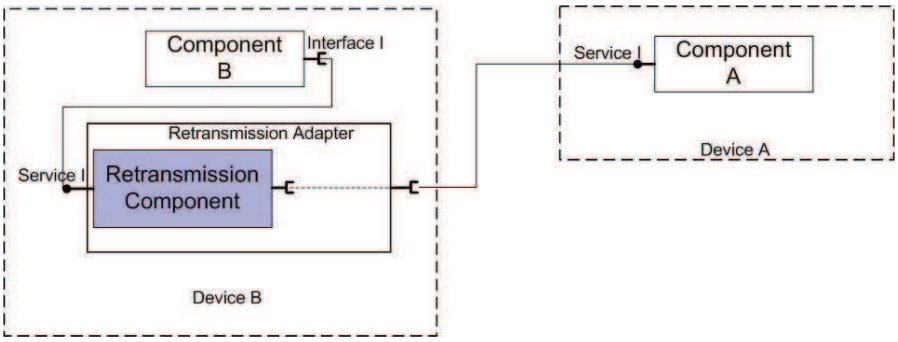


Figure 5.12: Retransmition adaptation pattern

Description This pattern provides the functionality of retransmitting a failed call to a remote component. Its adaptive logic component, as shown in the Figure 5.12, attempts to retransmit message to the target component.

Adaptation Context The transmission in pervasive environment may be subject to failures because of e.g. network down. This can be captured by tracking the network work for a period of time. If the statistics shows that the system is not reliable, thus, the components' messages could get lost along the path. When it is not possible to deliver messages to remote components, the system should attempt to respond to the component request at the earliest possible opportunity by trying to retransmit the messages. For this reason, we propose a retransmission pattern that attempts to retransmit the messages to render a user task reliable.

The retransmission pattern is used also during the execution of the task, if the network is quick cut-off, to overcome the loss of messages. Thus, once the network is repaired, the retransmission pattern is established to retry the sending of calls. To detect this adaptation context, we require to monitor the status of the supported network, i.e., if it is activated or not.

Where to use To ensure a reliable communication between devices, we propose to handle at init time a retransmission pattern in the sender side. For example, Figure 5.12 shows a retransmission pattern that is used by a device B to resend the failed call to a device A.

5.5.3 Example Scenario

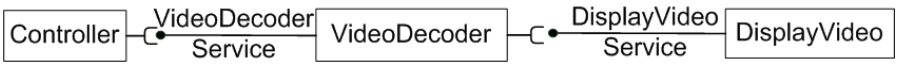


Figure 5.13: Abstract description of video player task

Referring back to the video player task described in Chapter 1 Figure 5.13 shows an abstract description of the task which consists of three abstract components namely, Controller, VideoDecoder and DisplayVideo services.

Assume that the controller component requires that its messages sent to the VideoDecoder service, should be authenticated. It expresses an authentication need towards the required service, i.e., VideoDecoder. However, the implementation of the concrete component of the videoDecoder service does not support the authentication intent. Thus, there is a mismatch between the given abstract description of the video player task and the concrete level. This gives a rise to the adaptation of the user task to fulfil its execution.

To resolve this mismatch, we can inject the authentication and integrity patterns into the abstract description of the task as shown in Figure 5.14. Thus, the controller component is able to send authenticated commands to the VideoDecoder component. These commands will be validated at first by the integrity pattern before forwarding it to the videoDecoder component. As a result, the task is transformed, as shown in Figure 5.14, to contain the authentication, and the integrity patterns in addition to its own components.

In another case, we assume that during the execution of the task that the bandwidth of the supported network becomes weak. This may have an impact on the quality of video

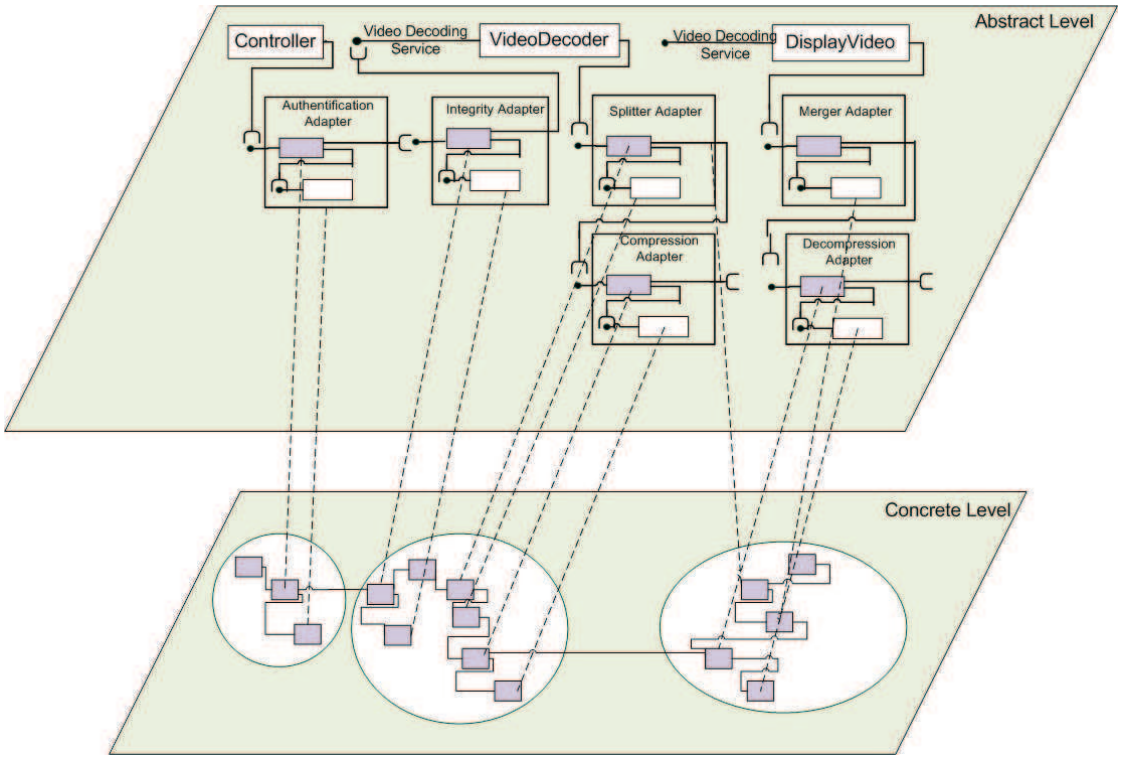


Figure 5.14: Adaptation of the video player task

that requires a high QoS. Towards this change of context, we can adapt the abstract video player task by splitting the frame into chunks and then compress them for a quick transfer.

For this purpose, we have composed together the splitting, compression, decompression and merging patterns, as shown in Figure 5.14 for a quick transfer of messages with a lower bandwidth. The splitter adapter will split a frame sent from the VideoDecoder component to the DisplayVideo one into chunks. These latter will be compressed before their transfer over the network. Once the chunks are received by a device, there is a need at first to decompress them and then to merge them before forwarding it to the DisplayVideo component. Hence, the abstract task is adapted by injecting a composition of adapters to overcome a mismatch triggered by a low bandwidth.

5.6 Conclusion

The adaptation of the user task may be triggered by the disappearance of a used device or component. It may be also caused by the appearance of a new device or the deployment of a new component that may be interesting for task's execution. Moreover, the changes of user preferences or devices capabilities may affect the devices values thus triggering the adaptation of the user task.

Towards these changes, we have proposed a compositional adaptation approach that is based on a partial reselection of services implementations as all components are not necessarily affected by the changes.

Moreover, the adaptation of a user task may be triggered by a mismatch captured between its abstract description and the concrete level, that denote that the user task can not be executed in the given context or in the new context if it changes. Hence, there is a need to adapt the abstract user task for a continuous execution.

Towards this objective, we have proposed a set of adaptation patterns that are injected into an abstract description of the task. These adapters exhibit an extra-functional behavior with respect to the functional behavior of the user task. The list of the adaptation patterns is not exhaustive. However, it is possible to define such other patterns following our adapter template, which consists of an adaptive logic compulsory component and extra-functional optional one.

Chapter 6

Middleware for the Continuity of User Tasks Execution

6.1 Introduction

We present in this chapter the MONitoring and ADAPTation (MonAdapt) middleware, which provides a comprehensive solution for the continuity of execution of user tasks in pervasive environments. This middleware integrates the contributions presented in this thesis, i.e., user tasks resolution (Chapter 3), monitoring of components (Chapter 4) and user tasks adaptation (Chapter 5).

The remainder of this chapter is structured as follows. First, we present in Section 6.2 the principle of Service Component Architecture (SCA), which is used to model the user tasks, and extended to meet the monitoring and adaptation needs. Then, we introduce in Section 6.3 the MonAdapt middleware architecture. Finally, we present a prototype implementation and performance evaluations of this middleware in Section 6.4.

6.2 User Tasks Description

6.2.1 Service Component Architecture

As the research in Service Oriented Architecture (SOA) is evolving, different architectures and languages have been proposed over the time to support application development using the SOA paradigm (Papazoglou, 2003). For example, Web Services (WS) have been particularly used for developing SOA-based applications. Unfortunately, due to its reliance on a particular set of description languages and protocols (WSDL/SOAP/HTTP), Web Services have suffered much from criticism.

Thus, a felt was needed for an architecture model that is independent of particular implementation technology or communication protocol. The result was in the form of the Service Component Architecture (SCA) (Open SOA Collaboration, 2007), which is maintained and standardized by the Open SOA consortium [1].

The main idea behind SCA is to be able to build distributed applications across organizations, which are independent of a particular technology, protocol, and implementation. Applications built as a set of services, called composite applications, can include both new services created specifically for the application, and also business functions from existing applications that are reused as part of the composition. SCA components are the basic units of construction of applications.

Figure 6.1 (a) shows the SCA meta-model, while Figure 6.1 (b) shows the various SCA elements that constitutes a composite in SCA.

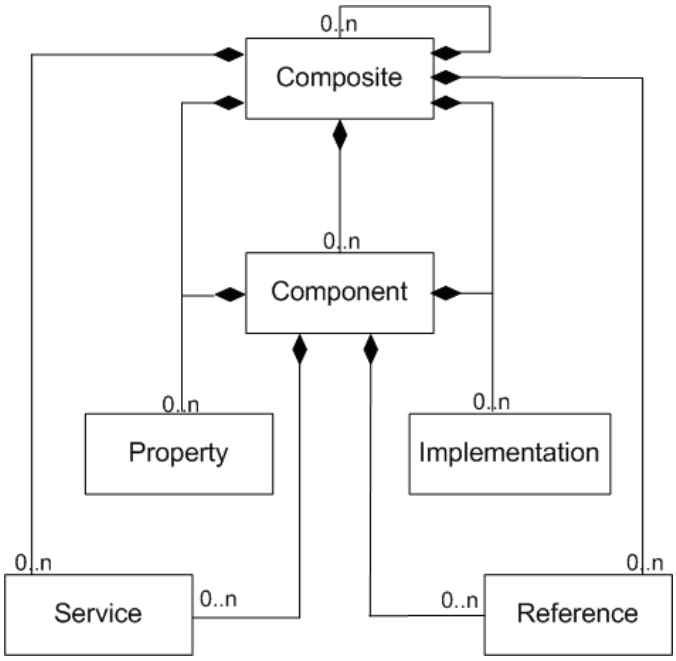
An SCA component encapsulates the implementation of a service and makes it available through clearly specified interfaces called SCA services. An SCA service is, thus, the access point to the functionality provided by the SCA component, i.e., its external interface. At the same time an SCA component expresses the dependencies on other services as SCA references that specify what a component needs from the other components or applications of the outside world. The SCA services and references used by a composite are exposed by promoting the interfaces of their internal SCA components.

Both SCA services and references are matched and connected using SCA wires that are specified using bindings. SCA wires are abstract representations of the relationship between references and some services that meet the needs of those references. The bindings specify how services and references communicate with each other. Each binding defines a particular protocol that can be used to communicate with a service as well as how to access it. A single service can have multiple bindings, allowing different remote software to communicate with it in different ways.

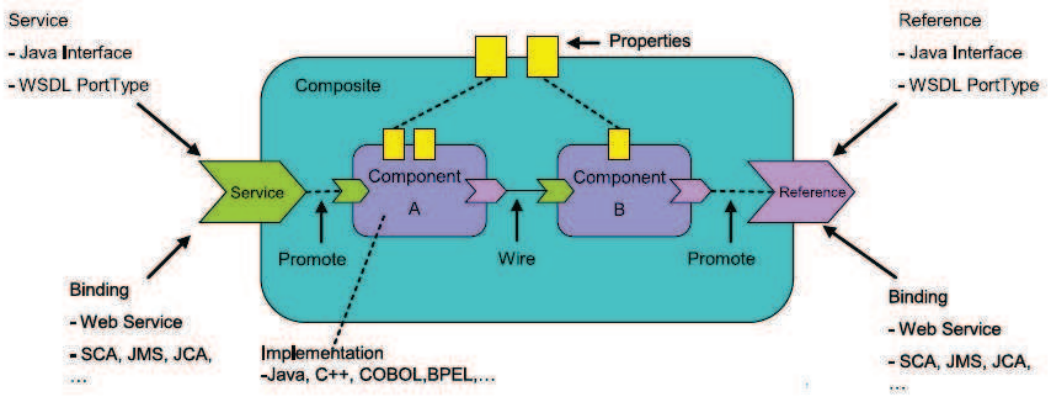
SCA components provide a mechanism to configure an implementation externally through SCA properties. These properties can be customized, allowing a component to adapt its behaviour appropriately. The implementation of an SCA component can be in Java, C++, COBOL, Web Services or as BPEL processes. Independent of whatever technology is used, every component relies on a common set of abstractions, i.e. services, references, properties, and bindings.

SCA divides up the steps in building a service-oriented application into two major parts:

1. http://www.osoa.org



(a) SCA meta model



(b) SCA composite

Figure 6.1: Service Component Architecture

- The implementation of components which provide services and consume other services
- The assembly of set of components to build business applications by connecting references to services.

SCA emphasizes the decoupling of service implementation and of service assembly from the details of infrastructure capabilities and from the details of the access methods used to invoke services. The assembly of an SCA application can be described using an XML-based ADL file, sometimes referred to as Service Component Definition Language (SCDL).

Different SCA Runtime implementations have been developed like IBM WebSphere Application Server [2], Apache Tuscany [3] and FraSCAti [4] (Seinturier et al., 2009), etc. An SCA Runtime provides the support to create, deploy and execute application based on the SCA specifications.

SCA Policy Framework

SCA provides a policy framework (Open SOA Collaboration, 2005) to support specification of non-functional constraints, capabilities, and QoS expectations from component design, which are captured and expressed through the introduction of policies. A policy describes some capability or constraint that can be applied to components or to the interaction between components. These policies are defined independently of the corresponding component assembly. The advantage is that policies can be reused for several different assemblies and can be changed without modifying the assembly itself. Thus, a separation is kept between functional and non-functional aspects of applications. The version 1.0 of the SCA Policy Framework discusses only the security and reliability policies.

SCA does not define how policies should be described within a domain —no policy language is mandated— so vendors are free to do this in their own ways. In order to allow the inclusion of any policy language within a policy set, SCA allows using the extensibility elements in the `@policySet` attribute, which may be from any namespace and may be intermixed. One or more policy sets can be attached to any SCA element used in the definition of components and composites.

In SCA, services and references can have policies applied to them that affect the form of the interaction that takes place at runtime. These are called interaction policies. Service components can also have other policies applied to them which affect how the components themselves behave within their runtime container. These are called implementation policies.

How a policy is interpreted depends on how the policy is defined within the domain in which the SCA component is running. For example, a binding for a service can have an associated policy set describing its interaction policies, while a binding for a reference can have another policy set describing its interaction policies. When a binding is created between them, these policy sets are matched, and their intersection determines the set of policies used for this communication.

2. http://www-01.ibm.com/software/webservers/appserv/was/featurepacks/sca/
3. http://tuscany.apache.org
4. http://frascati.ow2.org

```
<intent name="authentication" constrains="sca:binding">
<description>
Communication through this binding must be authenticated.
</description>
</intent>
```

Figure 6.2: SCA policy intent of an authentication requirement

The SCA policy framework defines the following key concepts:
- An *Intent* allows the SCA developer to specify abstract Quality of Service capabilities or requirements independent of their concrete realization.
- A *Profile* allows the SCA developer to express collections of abstract QoS intents.
- A *Policy Set* provides the realization of concrete policies for a set of intents.

For example, a service which requires its messages to be authenticated can be marked with an intent "authentication" as described in Figure 6.2. This marks the service as requiring message authentication capability without being prescriptive about how it is achieved. At deployment time, when the binding is chosen for the service (e.g., SOAP over HTTP), the deployer can apply suitable policies to the service which provide aspects of WS-Security, for example, and which supply a group of one or more authentication technologies.

Limitations of SCA

While SCA is gaining acceptance in the service-oriented industries, it has some limitations. First, the current SCA runtimes consider an SCA composite as a static assembly of components and cannot be modified during the execution of the application. The binding between the services and components is also defined at deployment time. Consequently any modification in the composition of the service-based application requires to stop the complete application, and restart it using the new composition. Second, the existing SCA specifications do no allow components to express their dependencies towards other components and leave this issue to the code level. Third, extra-functional aspects, like monitoring and dynamic reconfiguration, are not addressed by the current SCA specification. These aspects are usually left out of the specifications and must be handled by SCA platform implementations.

6.2.2 Extensions of SCA

In this thesis, we rely on the SCA specification to describe user tasks instead of defining a new component model. In this regard, SCA provides a rich ADL that details most of the aspects that we are looking for. One of the main features of this component model is that it is independent of a particular technology, a protocol, and an implementation. Using SCA, a user task is described as an assembly of services provided by various devices of the execution environment. We use SCA for its extensibility to overcome the missing

elements related to required properties, monitoring and adaptation aspects as shown in Figure 6.3.

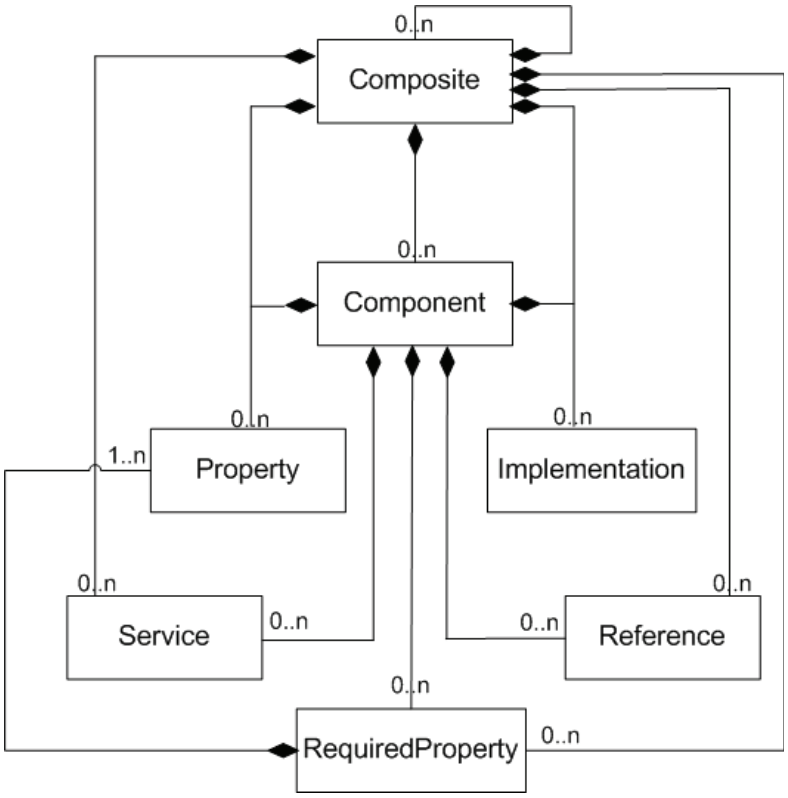


Figure 6.3: Extended SCA meta model

Modeling the behavior of a user task needs to satisfy not only the functional requirements in an effective way. But in order to provide better quality of service (QoS) for user satisfaction, it should also consider the current state of the environment in which the application is executing in order to be aware about its changes. In such situations, a component may depend on properties of components representing the environment or the user task.

Therefore, we have extended the standard SCA meta model, as shown in Figure 6.3, by adding `requiredProperty` element to express explicitly the dependency of components to the offered properties of other components (Belaïd et al., 2010). The component element can have zero or more `@requiredProperty` elements as children which are used to express that it requires certain property.

The `@requiredProperty` element has 1 or many properties to monitor or to reconfigure as shown in Figure 6.3. These properties may belong to hardware, software or network resource and they are specified by their names. The separation between required property and the associated property is significant as it allows the transformation of a given component for only the specified properties to be monitored or reconfigured (See Section 4.4).

```
<component ...>*
    <implementation ... />*
    <service ... />*
    <reference ... />*
    <requiredProperty resource="Name"
                      remotable="Boolean"
                      reconfiguration="Boolean"
                      monitoring="ByPolling|BySubscription"
                      notificationMode="OnChange|OnInterval"
                      startTime="long"
                      duration="long"
                      notificationInterval="long" >

        <property ... />*
        <property ... />*
        ...
    </requiredProperty>
</component>
```

Figure 6.4: Extension of SCA to support the monitoring and reconfiguration needs using required properties

The `@requiredProperty` element has the following attributes, as described in Figure 6.4.

- `resource` specifies the component to which this required property belongs. The resource attribute corresponds to classification of the component in one of the predefined categories, such as software, hardware, and network, etc. Some of these categories have been defined as extension of the Composite Capability/Preference Profile (CC/PP) (Kiss, 2007) by the authors (Mukhtar et al., 2008a) (Mukhtar et al., 2008b).
- `remotable` of SCA specification (Open SOA Collaboration, 2007) specifies if the required property belongs to a remote resource or not to carry out the remote monitoring or the remote configuration. This attribute has a boolean value; if it is true, it implies that the requested resource is remotable else it is a local resource. Its default value is false.
- `reconfiguration` specifies if the property of resource would be configured or not. This attribute has a boolean value; if it is true, it implies that the requested property has to be reconfigured. Its default value is false.
- `monitoring` is used to specify type of monitoring requirement; by polling or by subscription.
- `notificationMode` is used to distinguish between the monitoring by subscription modes. The monitoring by subscription mode *OnChange* specifies that the subscriber component is notified every time the value of the property changes. However, the monitoring by subscription mode *OnInterval* specifies that the subscriber component is to be notified after a specified time interval. Its default value is OnChange.

- `startTime` is used to specify the time period (in milliseconds) at which monitoring of the properties should start. If it is not specified, thus its default value is -1, and the subscription will be activated right after receiving the subscription message.
- `duration` indicates for how long (in milliseconds) the subscription should last. The default value of duration is -1, that means subscription all the time.
- `notificationInterval` is used to specify the time interval (in milliseconds) in which the notification is sent to the subscriber for a subscription mode *OnInterval*.

```
<component name="AdaptiveLogicOfAdaptationPattern" >*
    <service ... />*
    <reference ... />*
    <property ... />*
    <implementation.java generated="True"
                         type="Compression|Decompression|Proxy|Encryption
                         |Decryption|Authentication ..." />
</component>
```

Figure 6.5: Extension of Implementation element of a component

To cope with the adaptation issue, we have also extended SCA implementation element by a new attribute, `generated`, to specify if the implementation of the component is generated or not (Ben Lahmar et al., 2011a). This specification is useful to model the adaptation patterns for a structural adaptation. Indeed, the Adaptive Logic component of an adaptation pattern has a generated implementation because it depends on the business interfaces of components to be adapted in order to carry out the dynamic injection of adaptation patterns in users tasks.

Furthermore, a `type` attribute extends the implementation element of the Adaptive Logic component to determine whether this later is generated for an adaptation pattern allowing a compression or encryption or splitting, etc. Figure 6.5, shows the extension to the implementation child element of an SCA component.

6.2.3 Example Scenario

Task Description using Extended SCA

Referring back to the video player task in the Chapter 1. Figure 6.6 shows an extended SCA description of the video player task. The task is represented by a composite of the Controller, VideoDecoder and DisplayVideo components.

The Controller component requires that its messages sent to the *VideoDecoderService*, should be authenticated. Therefore, its reference is marked with an intent "authentication" (line 6 in Figure 6.6). However, the *VideoDecoderService* is marked with the "'integrity"' intent to check the validity of the messages received from the controller component (line 9 in Figure 6.6). Figure 6.2 shows a description of the authentication abstract intent that is applied to the component binding using SCA policy framework (Open SOA Collaboration, 2005).

```
1<composite name="VideoPlayer">
2  <component name="ControllerComponent">
3      <service name="ControllerService" requires="Hardware.Input">
4         <interface.java interface="example.interface.Controller" >
5      </service>
6      <reference name=" VideoDecoderService" requires="authentication"/>
7  </component>
8  <component name="VideoDecoderComponent" >
9      <service name="VideoDecoderService" requires="integrity">
10         <interface.java interface="example.interface.VideoDecoder" >
11     </service>
12     <reference name="DisplayVideoService" />
13  </component>
14  <component name="DisplayVideoComponent">
15     <service name="DisplayVideoService"
           requires="Hardware.Output.VideoCapable  Hardware.Output.SoundCapable">
16       <interface.java interface="example.interface.DisplayVideo" />
17     </service>
18  </component>
19 </composite>
```

Figure 6.6: SCA description of the video player task

```
1 <composite name="IntegrityAdapter">
2   <service name="GenericProxy" promote="AdaptiveLogicComponent/GenericProxyService"/>
3   <reference name=" VideoDecoderService" promote="AdaptiveLogicComponent/VideoDecoderService"/>
4   <component name="AdaptiveLogicComponent" >
5      <service name="GenericProxy">
6           <interface.java interface="mw.interface.GenericProxy"/>
7        </service>
8        <implementation.java  generated="True"  type ="Integrity" />
9      <reference name="IntegrityService" target="IntegrityComponent"/>
10     <reference name=" VideoDecoderService" target="VideoDecoderComponent" />
11  </component>
12  <component name="IntegrityComponent">
13       <service name="IntegrityService">
14      <interface.java interface="example.interface.Integrity" />
15       </service>
16    </component>
17 <composite>
```

Figure 6.7: SCA description of the integrity pattern

Based on the same concept, the services in the user task specify their requirements for capabilities abstractly. The component implementations provide the corresponding concrete policies. For example, the `ControllerService` specifies an intent *Hardware.Input* using the `@requires` attribute in the service definition, as shown in Figure 6.6, otherwise the service will not work properly. However, the DisplayVideoService requires *Hardware.Output.VideoCapable* and *Hardware.Output.SoundCapable* capabilities for its execution (line 15 in Figure 6.6).

Assume now that during the resolution of the task, the concrete components of the VideoDecoder and the controller services do not support the policies related to the authentication and integrity intents. This give a rise to an adaptation need in order to overcome the captured mismatch. Towards this challenge, it is possible to inject the authentication and integrity patterns into the user task to fulfil these missed requirements.

Figure 6.7 represents the SCA description of the integrity adaptation pattern. It consists of an AdaptiveLogic component and an IntegrityComponent one. The `@generated` attribute is used to specify that the implementation of the AdaptiveLogic component is generated (line 8). Moreover, the `@type` attribute (line 8) is used to specify the intent of the adaptation pattern, which is the checking the integrity of the received message from the Controller component. The authentication adaptation pattern is described in similar way.

```
1 <component name="Network.WiFi" >
2   <service name="WiFiService">
3      <interface.java
4       interface="example.interface.WiFiInterface"/>
5   </service>
6   <property name="signal" type="float"/>
7              . . .
8 </component>
```

Figure 6.8: SCA description of the WiFi component

```
1 <component name="Hardware.Battery" >
2   <service name="BatteryService">
3      <interface.java
4       interface="example.interface.BatteryInterface"/>
5   </service>
6   <property name="power" type="float" />
7              . . .
8 </component>
```

Figure 6.9: SCA description of the Battery component

As the task is executed in a heterogenous and dynamic environment, there is a need to monitor its changes and to carry out the necessary adaptation to fulfil the task execution.

```
1 <component name="TaskAdapter">
2   <service name="PCNotification">
3      <interface.java interface="mw.example.PCNotificationInterface"/>
4   </service>
5   <reference name="GenericProxy" target="Hardware.Battery"/>
6   <reference name="PCSubscription" target="Network.WiFi"/>
7   <requiredProperty resource="Hardware.Battery" monitoring="ByPolling">
8      <property name="power" type="float" />
9   </requiredProperty>
10  <requiredProperty resource="Newtork.WiFi" monitoring="BySubscription"
                                                notificationMode="ON_CHANGE" >
11     <property name="signal" type="float"/>
12  </requiredProperty>
13 <component>
```

Figure 6.10: SCA description of the TaskAdapter component

This is done by a TaskAdapter component that encapsulates the adaptation logic and allows the monitoring of properties which the task depend on. This component has to be defined by the architect at design time to make the task adaptable.

As shown in Figure 6.10, the TaskAdapter component expresses using its `requiredProperty` its needs to monitor *by polling* the `power` property of the Battery component belonging to a local hardware resource as described in Figure 6.9. It requires also to be notified about the changes of `signal` property of the WiFi component, belonging to the network category, each time it changes (see Figure 6.12). For this objective, a `monitoring` attribute is used to specify the subscription way for `signal` property. Moreover, a `notificationMode` attribute is used to specify the subscription mode that is OnChange. To receive the notifications, the TaskAdapter component provides a PCNotification service and has a reference to a PCSubscription service and a GenericProxy one.

Transformations to Standard SCA

Using the proposed monitoring and adaptation extensions provides the ability to an SCA component to specify its dependencies towards properties offered by other components and to monitor or to reconfigure them if there is a need. However, executing such extended descriptions is not possible in the existing opensource SCA runtimes [5] such as FraSCAti, Newton, or Tuscany, etc., as they do not support these monitoring and adaptation extensions. Therefore, to prove the feasibility and the efficiency of the proposed extensions, we proposed to transform them into standard SCA descriptions.

The transformation of the monitoring extensions into SCA standard corresponds to the creation of composites that encapsulate the concerned components, as specified using @resource attribute, with monitoring ones (see for more details in Chapter 4). In addition, the required properties extensions are transformed to references to the created composites.

Referring back to the video player task, as described in Figure 6.6, the TaskAdapter component expresses through its required properties its dependencies to the power prop-

5. http://osoa.org/display/Main/Implementation+Examples+and+Tools

```
1<composite name=BatteryComposite>
2  <service name="BatteryService" promote="Battery/BatteryService" />
3  <service name="GenericProxyService" promote="LGenericProxy/GenericProxy" />
6  <component name="Hardware.Battery" >
7      <service name="BatteryService">
8         <interface.java interface="example.interface.Battery"/>
9      </service>
10     <property name="power" type="float" />
11  </component>
12 <component name="LGenericProxy" >
13     <service name="GenericProxy">
14         <interface.java interface="mw.interface.GenericProxy"/>
15     </service>
16     <implementation class="mw.impl.LGenericProxy"/>
17     <reference name="BatteryService" target="Battery/BatteryService"/>
18  </component>
20</composite>
```

Figure 6.11: Transformation of the Battery component

```
1 <composite name="WiFiComposite">
2   <service name="WifiService" promote="WiFi/WifiService" />
3   <service name="SubscriptionService"
                promote="MonitorBySubscription/PCSubscription" />
4   <service name="GenericProxyService" promote="LGenericProxy/GenericProxy" />
5   <component name="Network.WiFi">
6      <service name="WifiService">
7         <interface.java interface="example.interface.Wifi"/>
8      </service>
9      <implementation class="example.Impl.WifiImpl"/>
10     <property name="signal"/>
11  </component>
12  <component name="LGenericProxy" >
13     <service name="GenericProxy">
14         <interface.java interface="mw.interface.GenericProxy"/>
15     </service>
16     <implementation class="mw.impl.LGenericProxy"/>
17     <reference name="target" target="Network.WiFi"/>
18  </component>
19  <component name="MonitorBySubscription" >
20      <service name="PCSubscription">
21         <interface.java interface="mw.interface.PCSubscription"/>
22      </service>
23      <implementation.java class="mw.impl.PCSubscriptionImpl"/>
24      <reference name="genericProxy" target="LGenericProxy/GenericProxy"/>
25   </component>
26 </composite>
```

Figure 6.12: Transformation of the WiFi component

erty of the Battery component and to signal property of the Wifi component. To consider these requirements using existing SCA runtimes, we propose to transform them into standard SCA descriptions.

In Figure 6.11, we show the transformation of the local Battery component to a new composite to render its property monitorable as requested by the TaskAdapter component. The created composite exposes in addition to the service of the Battery component, a GenericProxy service that is provided by a LGenericProxy component allowing the monitoring by polling of the power property.

Similarly we transform the WiFi component to render it monitorable as shown in Figure 6.12. The created composite exposes in addition to the service of the WiFi component, a GenericProxy service and a PCSubscription one.

```
1<component name="TaskAdapter">
2    ...
3      <reference name="GenericProxy" target="BatteryComposite"/>
4      <reference name="PCSubscription" target="WiFiComposite"/>
5   </component>
6   ...
7</component>
```

Figure 6.13: Transformation of the TaskAdapter component

```
1<composite name="TransformedVideoPlayer">
2  <component name="ControllerComponent">
3   . . .
4  </component>
5  <component name="VideoDecoderComponent" >
6   . . .
7  </component>
8  <component name="DisplayVideoComponent">
9   . . .
10 </component>
11 <component name="AuthenticationComponent">
12   <implementation.composite name="AuthenticationAdapter" />
13   . . .
14 </component>
15 <component name="IntegrityComponent">
16   <implementation.composite name="IntegrityAdapter" />
17   . . .
18 </component>
19 . . .
20 </composite>
```

Figure 6.14: SCA description of the transformed video player task

Moreover, the required properties of the VideoDecoder component are transformed to references to the created composites, as shown in Figure 6.13, namely, `BatteryComposite` and `WiFiComposite`.

To cope with the adaptation issue, we proposed to transform a user task to another one by injecting adaptation patterns that overcome the captured mismatches between its description and the concrete level.

The transformation of the video player task, as depicted in Figure 6.14, corresponds to the creation of a new composite that encapsulates in addition to the initial components of the task, the authentication and integrity composites.

6.3 MonAdapt Middleware

We present in this section, the middleware MonAdapt for task resolution, monitoring and task adaptation in pervasive environments. MonAdapt deals with the different contributions present in this thesis to ensure the continuity of user tasks execution in dynamic and heterogenous environments.

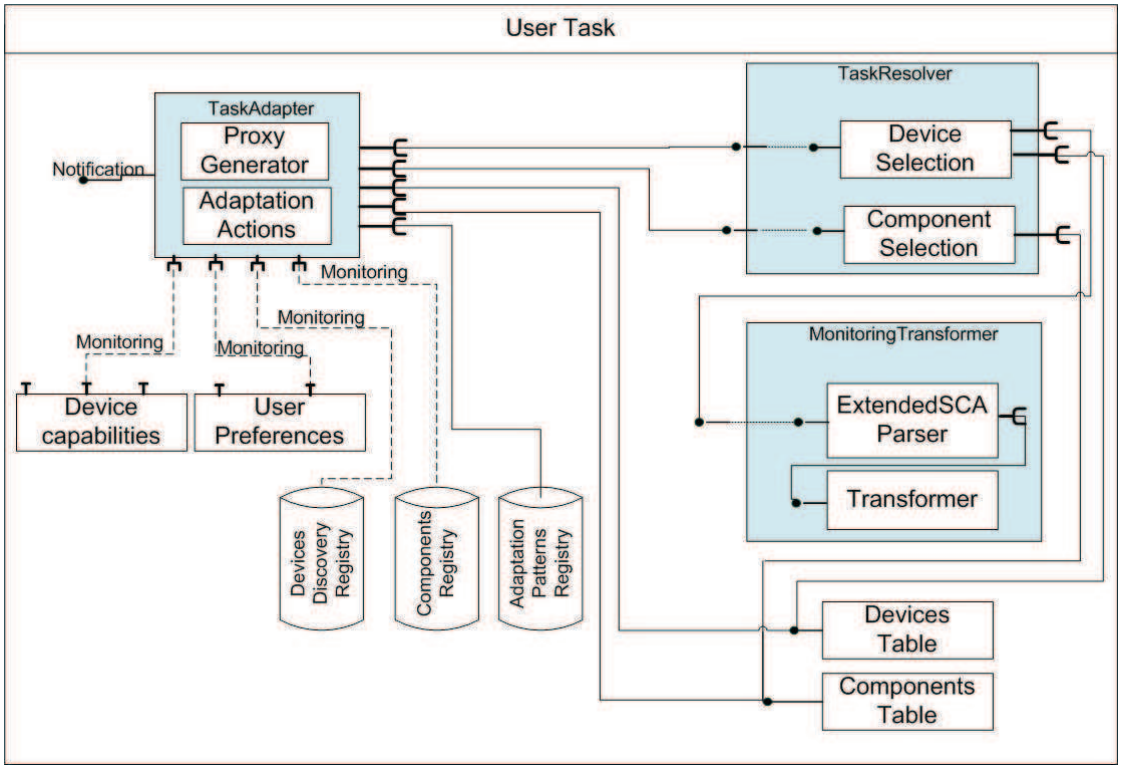


Figure 6.15: Architecture of MonAdapt Middleware

MonAdapt has been developed to consider user tasks described using SCA specification (Open SOA Collaboration, 2007). SCA allows user tasks to be defined in abstract way. Thus, task's services can be resolved by using concrete components provided by a variety of devices in a distributed fashion. MonAdapt takes an extended SCA description of a user task and transform it into a standard one and using the selection algorithms, it

finds the best solution for task execution in terms of devices and components available on the execution environment. Once a task is resolved, it may be subject to unforseen failure due to a change of user preference or device capability or appearance or disappearance of devices and components. In addition, some mismatches between its abstract description and the concrete level can be captured at init time or during its execution. Towards these changes, MonAdapt provides a monitoring and adaptation mechanism for a continuous execution of the user task.

The architectural design of our middleware is depicted in Figure 6.15. It consists of a TaskResolver, MonitoringTransformer and TaskAdapter components. The MonitoringTransformer is responsible for the transformation of extended SCA descriptions to standard ones and components to render them monitorable whenever there is a need. The TaskResolver component ensures the selection of the best device and component used for the execution of each service of the user task while the TaskAdapter adapts the execution of a user task for the new context.

The TaskResolver is based upon two components implementing the device selection and component selection algorithms (see Chapter 3 for details). The DeviceSelection component is executed by the user device that contains an abstract description of the task. For each user task, the middleware uses a DevicesTable component to update the devices lists associated to each service of the user task.

In addition, each selected device has a ComponentSelection component to match services with the best components. It uses the ComponentsTable component of the user task, to update the lists of components by adding or removing components associated with the services to execute in that device.

Moreover, each device in the execution environment contains a DeviceCapabilities component, which is a composition of software, hardware and network components to describe its characteristics. The DeviceCapabilities component provides properties corresponding to some of the exported properties of its internal components to represent its capabilities. The UserPreferences component is responsible for management of user preferences. The device also uses a ComponentRegistry that lists its deployed components and the DeviceRegistry component that is responsible for sending notifications about the arrival and departures of devices.

As these components have impact of the task's execution, there is a need to monitor their changes. This is done by the MonitoringTransformer component, that is based upon two components: ExtendedSCAParser and Transformer components. After parsing the extended SCA description by the ExtendedSCAParser component, the Transformer component transform them into monitorable composites and adds the notification codes to their implementations in order to send notifications to the subscribers components.

Once a change is captured, there is a need to adapt the task to ensure its continuous execution. This is carried out by the TaskAdapter component that is responsible for the decision making and the execution of adaptation actions. The TaskAdapter component deals with the partial reselection of services implementations and the structural adaptation to overcome failures and mismatches captured at init time or during the execution of the task.

To fulfil services reselection, it subscribes to the changes of the DeviceRegistry component and the remote ComponentRegistry components of the selected devices in order to receive notifications about the arrival and departures of devices or components. It

subscribes also to the changes of the UserPreferences and DeviceCapabilities components in order to be aware of the changes of their properties values.

The TaskAdapter component provides a notification service that allows it to receive notifications about the appearance, disappearance of devices or components, the changes of properties of user preferences or devices capabilities components. Given the received notification, it triggers the adaptation of the task to the new context by reselecting devices and/or components for a continuous execution of the user task. This is for, it requires the services offered by the DevicesTable and ComponentsTable components. In fact, the adaptation is carried out by its AdaptationAction component that implements the adaptation algorithms presented previously in Chapter 5.

An adaptation of the user task may be also triggered by mismatches captured at init time or during the execution of the user task. To overcome these mismatches, there is a need to restructure the task by injecting adaptation patterns that provide extra-functional behaviours. For this objective, the TaskAdapter consults AdaptationPatternsRegistry that contains descriptions of the proposed patterns. Then, it selects the suitable adaptation patterns to fulfil the resolution or the execution of the task. The TaskAdapter maintains a ProxyGenerator component that is responsible for the generation of the proxy implementation of the adaptation pattern. After that, it asks the TaskResolver to resolve the partial abstract description of the extra-functional component of the adaptation pattern to accomplish the injection of the adaptation pattern.

6.4 Prototype Implementation and Evaluation Results

In this section, we evaluate the performance of our MonAdapt middleware for the three aspects namely, task resolution, monitoring and task adaptation. We begin by detailing the implementation of our prototype and the experimentation setup. Then, we evaluate the time required to fulfil the device selection and component selection for a task resolution. After that, we evaluate the transformation of a component in order to render it monitorable. Finally, we evaluated the time required for a generation of a proxy component for a structural adaptation of a user task.

6.4.1 Implementation and Experimentation setup

The MonAdapt middleware is described as an SCA composite. Each component has a java implementation. The TaskResolver composite consists of two components namely, DeviceSelection and ComponentSelection components, that implements the resolution Algorithms (as detailed in Chapter 3) using Java version 1.6.

The MonitoringTransformer component is composed of an ExtendedSCAParser and a Transformer components. The Transformer component is responsible for adding notifications code to components to render them monitorable. To that end, it uses the java API **java.lang.reflect** to obtain information about classes and objects. It is also based on the open source software **Java programming ASSISTant** [6] **(JavAssist)** library that is used to enable Java programs to define new classes at runtime or to modify a class byte code.

6. http://http://www.csg.is.titech.ac.jp/

Moreover, the LGenericProxy component is also responsible for transforming a component to a monitorable composite, as detailed in Chapter 4. The LGenericProxy component relies on **java.lang.reflect** to obtain information about the monitored component and to invoke the methods on it.

For remote monitoring, two transformations are handled in the server and in the client sides. To that end, we have used the **Universal Plug and Play (UPnP)** (UPnP Forum, 2008) technology to allow devices to exchange their capabilities or to discover one another and components to monitor remote properties. A UPnP network consists of UPnP devices that act as servers to UPnP control points. The control point can search for devices and invoke actions on them.

The remote monitoring in the server side is carried out by an RServer component that integrates the network communication aspects like remote method call and event processing. RServer component is implemented as a UPnP device, using **CyberLink library**[7], to offer UPnP services and UPnP device description to remote clients.

In the client side, the RGenericProxy component, providing a remote generic proxy service, is implemented as an UPnP control point using CyberLink library. When it starts, it searches for a UPnP device with the same type of RServer component and subscribes to the UPnP events related to the change of the variables state of this server component. In addition, the implementation of the ServerProxy component is generated using the JavAssist library to represent the remote component.

To fulfil the adaptation, MonAdapt provides a TaskAdapter component that implements the adaptation Algorithms as detailed in Chapter 5. This component contains a GeneratorProxy component to generate the byte codes of the proxy components of the adaptive logic of an adaptation patterns by using JavAssist.

The evaluation was carried out using a simulated environment on a single PC consisting of 2.27 GHz dual processor with 2GB RAM. The simulation consisted of running several devices on the same PC. Each device's capabilities were represented by CC/PP profile (Kiss, 2007) in terms of screen sizes, type of input, etc. We have specified 30 capabilities based on CC/PP profile of existing devices.

A device is considered as an SCA component whose properties correspond to its capabilities. We have generated randomly 100 devices by varying their properties values towards these capabilities. A user is also represented as an SCA component. Similarly, a user has properties values towards the specified devices capabilities.

Then, we have generated a random task with a random number of abstract components. Each abstract component has a different set of services, references, properties. An abstract component may have requirements towards the capabilities of devices. We have specified 15 services requirements. Each abstract component may have from 0 to 15 requirements towards the capabilities of devices. Each service or reference of a component has an interface that contains a random number of methods. A method of an interface has a random return type and a random number of parameters.

6.4.2 Evaluation of Task Resolution

In this section, we evaluate the time required for the device and component selections to fulfil an abstract component resolution of a user task.

7. http://www.cybergarage.org/twiki/bin/view/Main/CyberLinkForJava

Device Selection We evaluated the time required for the device selection of an abstract component of the user task by varying the number of the devices and the services requirements. We denote that the device selection considers the user preferences, devices capabilities and services requirements.

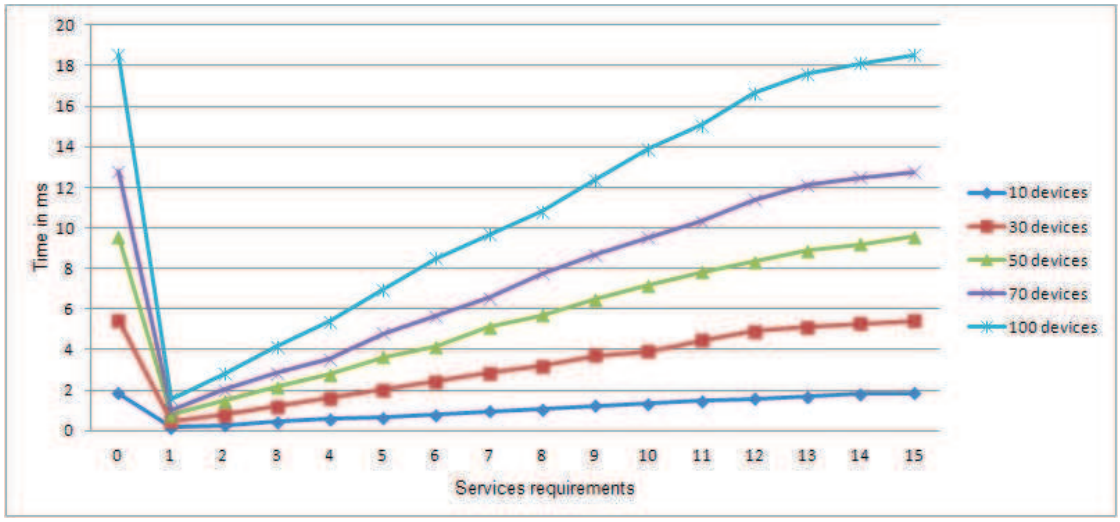


Figure 6.16: Device selection for an abstract component by varying its requirements and the number of devices

We have varied the number of devices of the execution environment between 10 and 100 and the services requirements of the abstract component between 0 and 15 in order to calculate the time required for the selection of a device for an abstract component of a task. Each device maintains a concrete component matching with the abstract component to valid the fitness of the abstract component in all available devices of the execution environment.

Figure 6.16 shows the time measured to fulfil the device selection for an abstract component of a user task by varying its requirements towards the devices capabilities and the number of devices. As it can be seen, the time required for a device selection among different devices of the execution environment is linear. We also notice that the device selection time is in the order of few milliseconds. For example, if the environment contains 30 devices and an abstract component has 10 services requirements, the device selection time takes around 4 ms.

This time depends also on the number of services requirements. We have varied the number of services requirements between 0 and 15. The Figure 6.16 shows that the time is as few as the number of services requirements that are considered for a device selection. For example, selecting a device among 100 alternatives to execute an abstract component having 1 requirement is around 0.3 ms, while this time takes more than 14 ms if the abstract component expresses more than 10 requirements.

The time required for the selection of a device using the algorithm in (Mukhtar et al., 2009) corresponds to 0 requirements in Figure 6.16. As it can be seen there are equivalent results for the selection of a device for the abstract component with 0 or 15 requirements as the 15 requirements include all the devices capabilities. The Figure 6.16 also shows that the variability of the number of services requirements between 1 and 15 has an impact

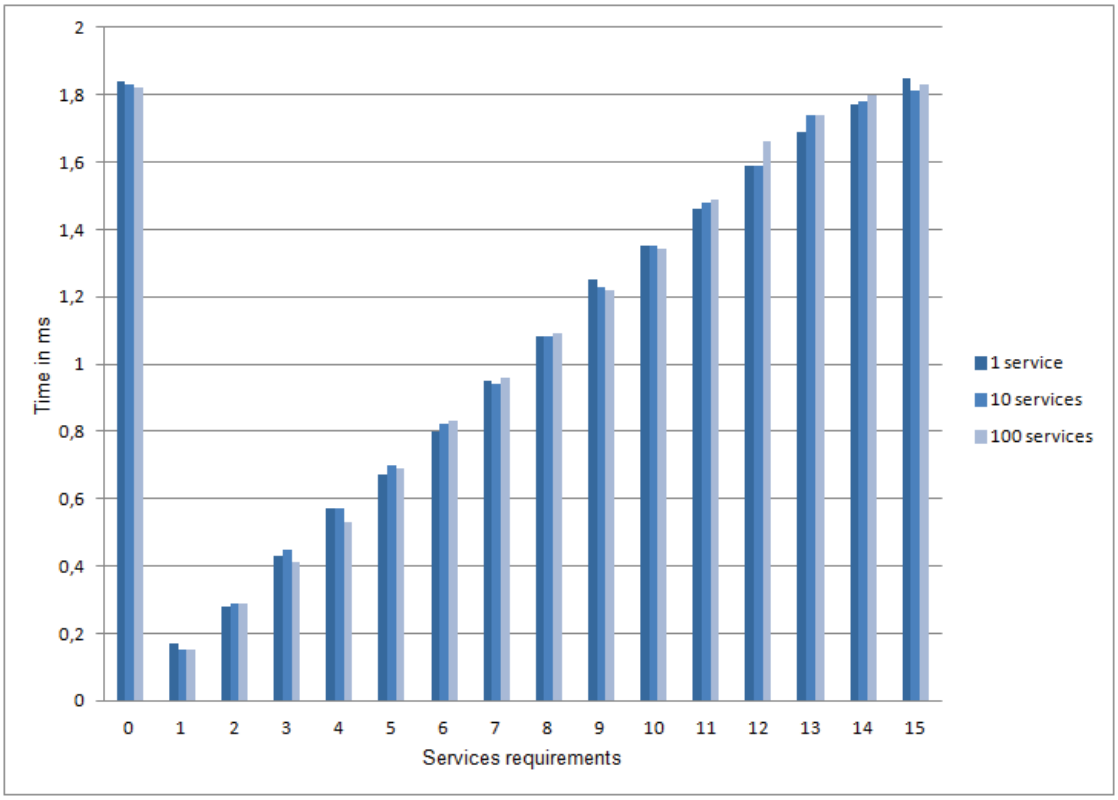


Figure 6.17: Device selection for an abstract component by varying its requirements and its offered services

on the device selection time that is fewer than neglecting the services requirements (i.e., 0 requirements). This proves that our device selection algorithm improves the algorithm in (Mukhtar et al., 2009) and it provides best results when considering the requirements of abstract components towards the device capabilities.

We also studied the impact of the variability of the number of services offered by an abstract component for the time required for a device selection, as shown in Figure 6.17. For this, we have varied the number of services offered by an abstract component between 1 and 100 and its requirements between 0 and 15. The execution environment contains 10 devices.

As it can be seen in Figure 6.17, increasing the number of services offered by an abstract component does not influence the time required for the device selection. However, this experiment confirms another time the dependency of the device selection to the number of requirements of the abstract component.

Component Selection To fulfil a resolution of an abstract component, there is a need to match it with a concrete component in the selected device. We varied the number of concrete components in the selected device and their preferences in order to measure the time required for the component selection for an abstract component. The number of concrete components in each device is varied between 1 and 100, the components' preferences are varied between 0 and 30.

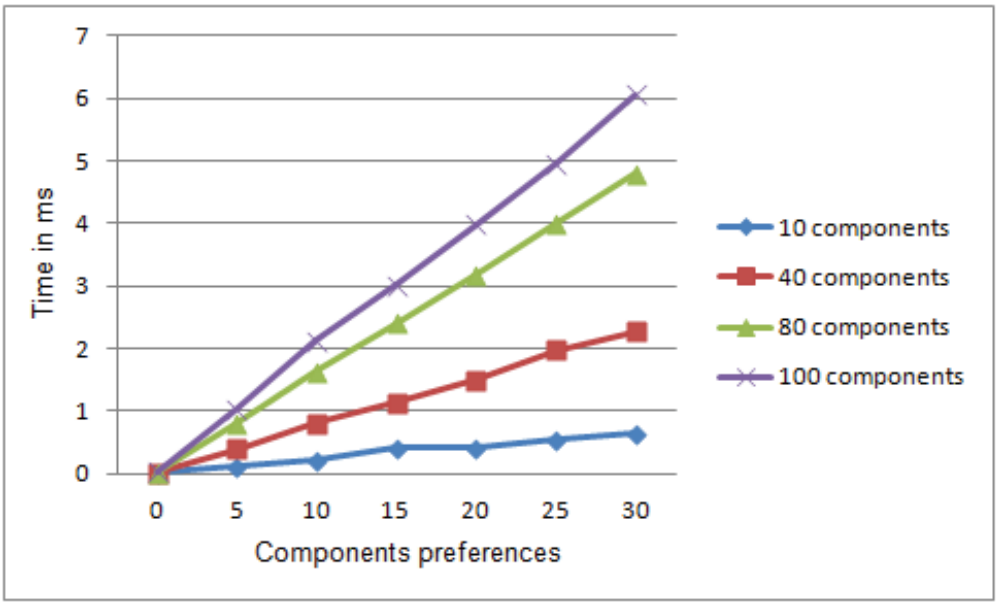


Figure 6.18: Component selection for an abstract component by varying the matching concrete components and their corresponding preferences

As shown in Figure 6.18, the selection of a component depends on the number of concrete components matching with the service in the selected device. Thus, increasing the number of components in the selected device implies an augmentation of the time of the component selection.

The time of component selection is also dependent on the number of components preferences considered for the component selection. For example, the selection of suitable component among 10 alternatives whose their preferences is about 20, takes less than 1 ms.

6.4.3 Evaluation of Monitoring

For the monitoring aspect, we are interested in evaluating the transformation of a component to render it monitorable. The monitoring transformation consists of adding, to its byte code at runtime, notification codes to the services methods and to the accessors and mutuators of properties. For this objective, we compared the impact of the variability of the number of properties and methods on the time required for a component transformation. For the following experiments, we consider that the component implements one interface.

Figure 6.19 compares the effect of each factor on the time of transformation. To measure the impact of properties, we vary the number of properties between 1 and 200, while the number of methods of the implemented interface is fixed to 1. However, to calculate the effect of the number of methods, we vary the number of the interface's methods between 1 and 200 and we fixe the number of properties to 1.

As it can be seen in Figure 6.19, the time of transformation of a component for a monitoring is linear. It is related to the number of its properties and to the number

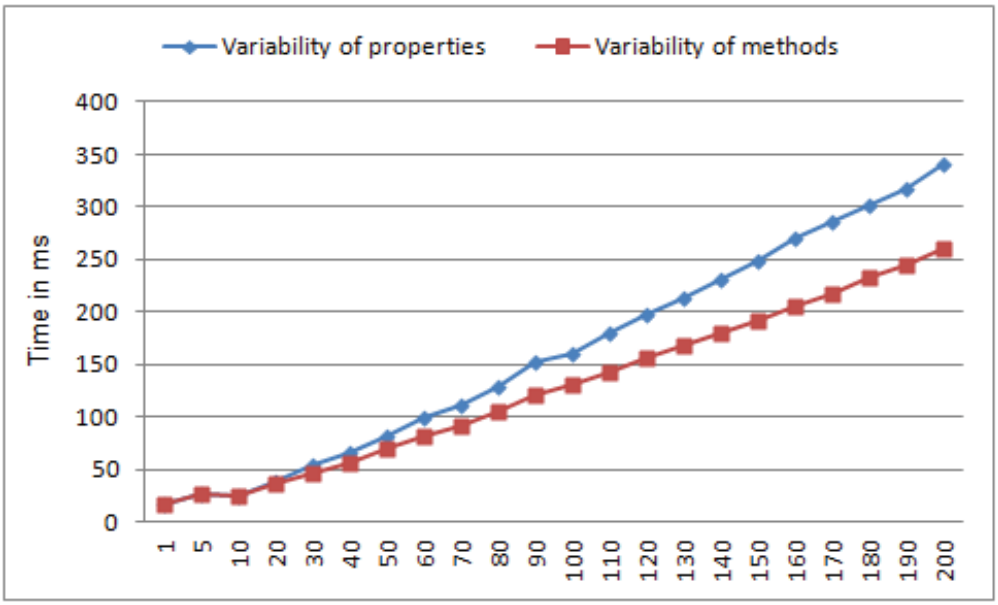


Figure 6.19: The effect of the variability of the number of the properties or methods on the transformation of a component for a monitoring need

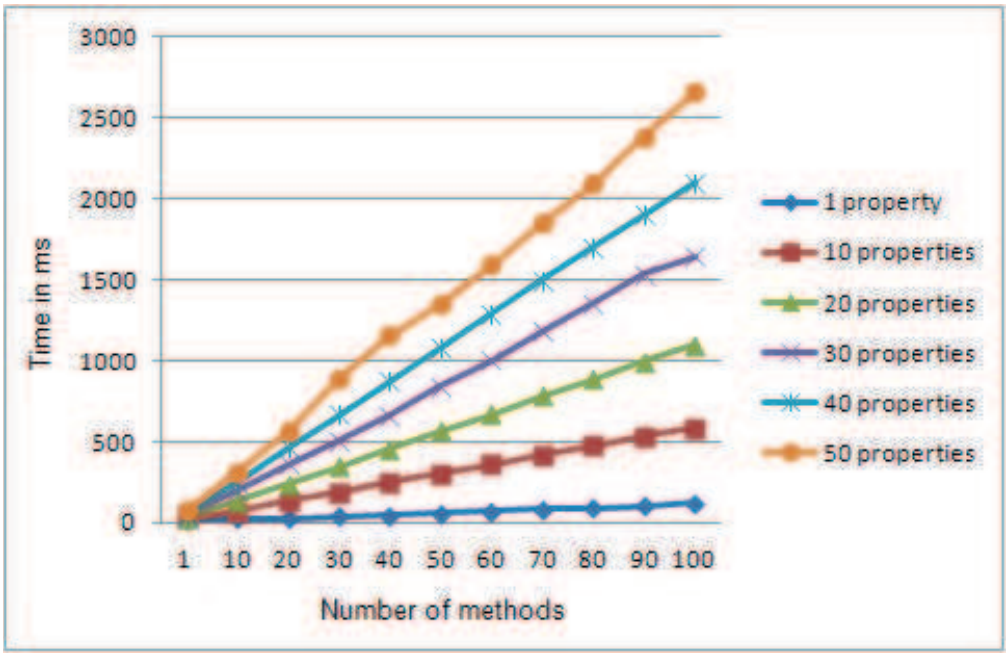


Figure 6.20: Transformation of a component for a monitoring by varying the both number of its properties and the methods

methods of the implemented interface. This is due to the injected notification codes that depend on the number of monitored properties. If the number of properties is high, the byte code of notifications increases too. We also notice that the time for a transformation is in the order of few milliseconds. For example, it takes around 25 ms if the component exposes 10 properties and an interface with only method or 1 property and an interface with 10 methods.

To confirm the influence of the both factors, we have varied the number of methods between 1 and 100 and properties between 10 and 50 as shown in Figure 6.20. As it can be seen, expanding the number of offered properties has a big impact on the time required for transformation of a component than increasing the number of the interface's methods. For example, the transformation a component having 20 properties represents the double of the time to transform a component having 10 properties.

6.4.4 Evaluation of Task Adaptation

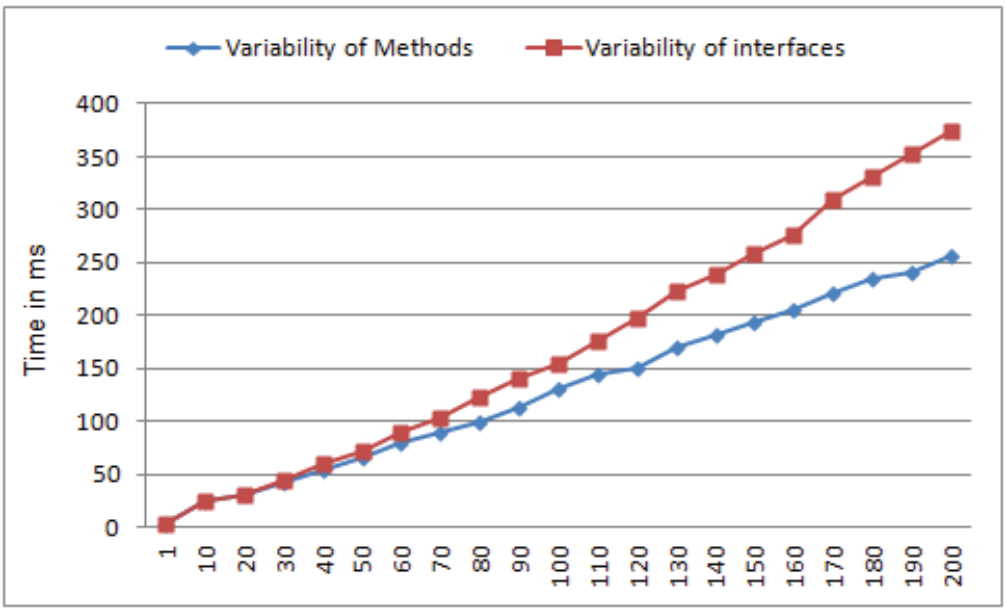


Figure 6.21: The effect of the variability of the number the methods and interfaces on the generation of a proxy component

In this section, we do not consider the adaptation contexts that trigger the reselection of services' implementation as this later refers to the selection of a component and/or device, which was evaluated previously in Section 6.4.2. However, we are interested in knowing how quickly the structural adaptation takes place to cover the task's mismatches. Irrespective of the situation and the used pattern, we need to generate the byte code of its adaptive logic component as explained in Chapter 5. This generated component corresponds to a proxy that implements some business interfaces. Therefore, we have evaluated the structural adaptation by calculating only the time required for the generation of the byte code of this proxy.

For this experiment, we varied the number of methods and interfaces between 1 to 200, to deduce which aspect has an important impact on the time required for the generation of the proxy component. To observe the impact of methods' variability, we fixed the number of interfaces to 1. However, the number of methods is fixed to 1 to evaluate the impact of the variability of number of interfaces on the proxy generation's time.

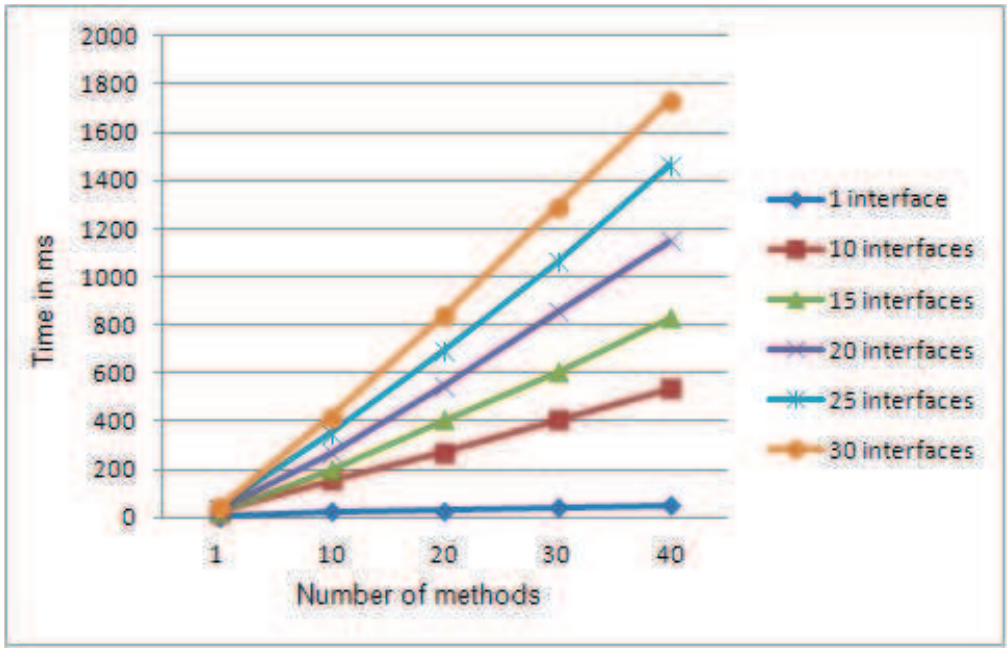


Figure 6.22: Generation of a proxy component by varying the both number of the methods and the interfaces that it implements

We deduce from the Figure 6.21, that the variability of interfaces has more impact on the time needed for the generation of the byte code of a proxy than the variability of methods of an interface. Indeed, the generation of a proxy consists of the time for loading interfaces classes from memory and the time of byte code class generation. Therefore, increasing the number of interfaces implies the augmentation of the time of proxy's generation.

We have also evaluated the variability of interfaces and methods to observe the influence of the both factors on the time required for the generation of proxy. The number of interfaces is varied between 1 and 30, whereas, the number of interfaces is varied between 1 and 100, as shown in Figure 6.22.

We observe that the time of generation of the proxy is linear and it is in the order of some milliseconds. For example, the generation of a proxy with 10 interfaces having 10 methods takes less than 200 ms.

6.5 Conclusion

In this chapter, we proposed some extensions to SCA specification that are used for describing the users tasks in abstract way. These extensions deal mainly with the

monitoring and adaptation aspects in order to allow an SCA-based component to express its dependencies to external properties.

We then presented the architecture and functionalities of our proposed middleware MonAdapt for task resolution, monitoring the changes of the underlying environment and task adaptation. We provided some implementation details of the middleware's components along with evaluation results for times required for an abstract component's resolution, the transformation of a component to render it monitorable and the generation of proxy component in case of a structural adaptation.

Chapter 7

Conclusion

The emergence of wireless technologies and ubiquity of hand held wireless devices has increasingly enabled the pervasive computing that envisions the seamless applications. In pervasive environments, applications are cooperatively executed by integrating transparently functionalities provided by heterogeneous software and hardware resources in order to assist users in the realization of their daily tasks. Towards this evolution, Service Oriented Architecture (SOA) has emerged as a computing paradigm that changed the traditional way of how applications are designed, implemented and consumed in a pervasive environment.

Building upon the SOA, it is possible to describe a user task as an assembly of abstract components (i.e. services), which are reusable software entities with well defined interfaces, and may be accessed without any knowledge about their implementations or the programming languages. Thus, a user task can be crafted using a set of abstract components requiring and providing services to/from one another. This allows the separation of the business functionality (services) from its implementation (concrete components) and to execute a user task by composing different components provided by various devices.

The implementation of these services can be found by looking up concrete and deployed components in devices of the environment, which implies the resolution of abstract components into concrete ones. A service is matched with a concrete component if their interfaces match. A user task is said to be resolved if for all of its abstract components, we find matching concrete components implementing these services.

However, resolving an abstract description of a user task can be done effortlessly. In such heterogenous environments, we may find several matching components across different devices, which offer similar functional interfaces. So there is a need to a mechanism that selects for each service of the task the suitable concrete component for its execution. Furthermore, tasks in pervasive environments are challenged by the dynamism of their execution environment due to, e.g., user and device mobility. This arises the need to monitor the changes of the environment, which may make them subjects to unforeseen failures. Towards these captured failures, a dynamic adaptation is required to fulfill its execution in an ever changing and dynamic environments. These problems put the light

on a relevant challenge, which is ensuring the continuity of execution of user tasks in such dynamic and heterogeneous environments.

7.1 Contributions

In this thesis, we focus our interest upon the challenge of the continuity of user tasks executions in pervasive environments. This challenge is divided into three subproblems namely, task resolution, monitoring of pervasive environments and task adaptation. If many middleware dealt with one or more of our service problems - composition, monitoring, adaptation - few proposed a unified vision for the continuity of applications executions in such dynamic and heterogeneous environments. This thesis contributed to these three aspects. Our contributions in this regard can be summarized in the following.

Apart from an abstract description of a user task, we propose to resolve it to fulfil its execution. The resolution of a user task involves an automatic selection of concrete components across various devices in the environment. Towards this objective, we have considered in addition to the functional aspects of a task, some non-functional ones like user preferences, devices capabilities, services requirements and components preferences. These non-functional aspects allow, for each service of the user task, the selection of a suitable component in a convenient device among a large list of devices and components provided by the execution environment.

Due to the heterogeneity and dynamicity of pervasive environment, an important aspect of user tasks is that their execution is very much dependent on their context. Therefore, modeling the behavior of a user task needs to satisfy not only the functional requirements in an effective way, but in order to provide better quality of service (QoS) for user satisfaction, it should also consider the current state of the environment in which the application is executing. For this, we have extended the specification of the Service Component Architecture (SCA) to allow a component to express explicitly its dependencies to external properties in terms of required properties, which are offered by other components.

Using the required properties, a component may specify its monitoring needs in order to be aware about the changes of the properties offered by local or remote components. We distinguish two monitoring types: by polling and by subscription. Polling is the simpler way of monitoring, as it allows the observer to request the current state of an external property whenever there is a need. However, subscription allows an observing component to be notified about changes of monitored properties. There are two modes of monitoring by subscription: 1) subscription OnChange which specifies that the subscriber component is notified every time the value of the property changes; 2) subscription OnInterval which specifies that the subscriber component is to be notified after a specified time interval.

We proposed also some transformation mechanisms to render local as well as remote components monitorable, if they are not, in order to respond to components requests. A monitoring transformation consists of creating a composite that encapsulate the concerned component with defined ones. The created composite provides in addition to the component's services, the monitoring ones through which it allows other components to subscribe or to observe the changes of its properties.

Moreover, user tasks in pervasive environments are challenged by the dynamicity of their execution environments due to, e.g., users and devices mobility, which make them

subject to unforeseen failures. We distinguish two major events categories triggering the adaptation of user tasks in pervasive environments namely, failures and mismatching. The failures may be caused at runtime by changes of user preferences, devices capabilities, appearance/disappearance of devices or components and so forth. However, the mismatching between abstract and concrete levels can be captured at init time or during the execution of the task to denote the inability of the abstract description to be executed in the given context or in the new context if it changes. Towards the captured failures or mismatches, there is a crucial need to overcome them and thus to adapt the user tasks to ensure the continuity of their execution.

To overcome the captured failures, we proposed a compositional adaptation approach that is based on a partial reselection of devices and components as in many cases there are a lot of devices and components that are not affected by the changes. The main objective of this approach is to ensure of each service of the task to overcome the captured changes by selecting the suitable device and component that maximizes the user and components preferences and that satisfies the services requirements regarding the existing devices capabilities.

In case of captured mismatches, we require to restructure the task's description with respect to its functional behaviour, so that it can be resolved and executed using the available resources of the execution environments. This is done by adding adaptation patterns that are injected into an abstract description of the task. These adapters exhibit an extra-functional behavior with respect to the functional behavior of the application and they are defined following our adapter template, which consists of an adaptive logic compulsory component and extra-functional optional one. We have identified a list of adaptation patterns that allows encryption, decryption, compression, splitting, etc. The list of the adaptation patterns is not exhaustive. However, it is possible to define some other patterns following our adapter template.

To have a unified vision of these contributions, we proposed an architectural design of a middleware allowing the task's resolution, the monitoring of the environment and the task adaptation. We provide implementation details of the middleware's components along with evaluation results to prove the feasibility and the efficiency of our proposed approaches.

7.2 Perspectives

We can distinguish tow kinds of perspectives, the short term perspectives which are related to improving our middleware with additional functionalities, and the long term perspectives which can be realized after sufficient time for investigation before they can be solved.

Among the short term perspective, the most important aspect to consider is the specification of adaptation policies to describe when and which adaptation patterns will be injected dynamically to fulfill the execution of the task. These policies are of the form of event-conditions-actions. While events in the environment correspond to the captured mismatches, actions consist of identification of adaptation patterns that will be used separately or composed together to restructure a task's description.

Another short term perspective that can be investigated is to extend our middleware with de-adaptation functionality. This is required when the context is recovered after

adaptations actions. So there is no need to the injected adaptation patterns. For example, if the compression and decompression patterns are used to overcome a lower bandwidth, and this latter is repaired, we require to remove these injected patterns to re-establish the direct interaction between the concerned components rather than adapting it.

As long term objectives, we envision to integrate our prototype into an existing SCA runtime, like Frascati [1] or Newton [2], to support the proposed extensions for monitoring and adaptation features. This will allow us to test the feasibility of our approach in real world scenarios.

Furthermore, our middleware deals actually with stateless services for the task resolution and adaptation. In the future, we would like to extend it with functionality allowing the reselection of stateful components, which require not only the identification of the replacing components but also to transfer their states.

1. http://frascati.ow2.org
2. http://newton.codecauldron.org

Bibliography

Abowd, G. D., Dey, A. K., Brown, P. J., Davies, N., Smith, M., and Steggles, P. (1999). Towards a better understanding of context and context-awareness. In *the 1st international symposium on Handheld and Ubiquitous Computing*, HUC' 99, pages 304–307, Karlsruhe, Germany.

Baldoni, R. and Virgillito, A. (2005). Distributed event routing in publish/subscribe communication systems: a survey. Technical Report 15-05, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienzia", Rome, Italy.

Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, V., Henrio, L., and Pérez, C. (2009). Gcm: a grid extension to fractal for autonomous distributed components. *Annales des Télécommunications*, 64(1-2):5–24.

Becker, C., Handte, M., Schiele, G., and Rothermel, K. (2004a). Pcom - a component system for pervasive computing. In *the Second Conference on Pervasive Computing and Communications*, PerCom 2004, pages 67–76, Orlando, FL, USA.

Becker, S., Brogi, A., Gorton, I., Overhage, S., Romanovsky, A., and Tivoli, M. (2004b). Towards an engineering approach to component adaptation. In *Architecting Systems with Trustworthy Components*, pages 193–215.

Belaïd, D., Ben Lahmar, I., and Mukhtar, H. (Dec 2010). A Framework for Monitoring and Reconfiguration of Components Using Dynamic Transformation. *International Journal On Advances in Software*, vol 3 no 3&4:371 – 384.

Ben Lahmar, I., Belaïd, D., and Mukhtar, H. (2010a). Adapting abstract component applications using adaptation patterns. In *Proceedings of the Second International Conference on Adaptive and Self-adaptive Systems and Applications*, Adaptive'10, pages 170–175, Lisbon, Portugal.

Ben Lahmar, I., Belaïd, D., and Mukhtar, H. (2011a). A Pattern-based Adaptation for Abstract Applications in Pervasive Environments. *International Journal On Advances in Software*, vol 3 no 3&4:367 – 377.

Ben Lahmar, I., Belaïd, D., and Mukhtar, H. (2012). Middleware for task resolution and adaptation in pervasive environments. *Evolving Systems Springer Journal*, 3.

Ben Lahmar, I., Belaïd, D., Mukhtar, H., and Chaudhary, S. (2011b). Automatic task resolution and adaptation in pervasive environments. In *the Second International Conference on Adaptive and Intelligent Systems*, ICAIS' 11, pages 131–144, Klagenfurt, Austria.

Ben Lahmar, I., Mukhtar, H., and Belaïd, D. (2010b). Monitoring of non-functional requirements using dynamic transformation of components. In *Proceedings of the*

6th International Conference on Networking and Services, ICNS'10, pages 61–66, Cancun, Mexico.

Ben Mokhtar, S., Georgantas, N., and Issarny, V. (2007). COCOA: COnversation-based service COmposition in pervAsive computing environments with QoS support. *Journal of Systems and Software*, 80(12):1941–1955.

Benatallah, B., Casati, F., Grigori, D., Nezhad, H. R. M., and Toumani, F. (2005). Developing adapters for web services integration. In *In Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE), Porto,Portugal*, pages 415–429. Springer Verlag.

Beugnard, A., Chabridon, S., Conan, D., Taconet, C., Dagnat, F., and Kaboré, E. (2009). Towards context-aware components. In *Proceedings of the first international workshop on Context-aware software technology and applications*, CASTA '09, pages 1–4, Amsterdam, Netherlands.

Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D. (2004). Web Services Architecture. *W3C Working Group Note*, 11:2005–1.

Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Software Practice and Experience (SP&E)*, 36:1257–1284.

Burbeck, S. (2000). The Tao of e-business services: The evolution of Web applications into service-oriented components with Web services. http://www-106.ibm.com/developerworks/Webservices/library/ws-tao/.

Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. (2001). Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383.

Computing, A., Paper, W., and Edition, T. (2006). An architectural blueprint for autonomic computing. *Quality*, 36(June):34.

Conan, D., Rouvoy, R., and Seinturier, L. (2007). Scalable Processing of Context Information with COSMOS. In *7th IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 210–224, Paphos, Chypre.

Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., and Ueyama, J. (2004). A component model for building systems software. In *IASTED Software Engineering and Applications*.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131.

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

Fuentes, L., Gámez, N., Pinto, M., and Valenzuela, J. A. (2007). Using connectors to model crosscutting influences in software architectures. In *ECSA*, pages 292–295.

Fujii, K. and Suda, T. (2005). Semantics-based dynamic service composition. *IEEE Journal on Selected Areas in Communications*, 23(12):2361–2372.

Fujii, K. and Suda, T. (2009). Semantics-based context-aware dynamic service composition. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–31.

Galster, M. and Bucherer, E. (2008). A taxonomy for identifying and specifying non-functional requirements in service-oriented development. In *IEEE Congress on Services*, SERVICES '08, pages 345–352, Honolulu, Hawaii, USA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Grace, P., Blair, G. S., and Samuel, S. (2003). Remmoc: A reflective middleware to support mobile client interoperability. In *CoopIS/DOA/ODBASE*, pages 1170–1187.

Handte, M., Herrmann, K., Schiele, G., and Becker, C. (2007). Supporting pluggable configuration algorithms in pcom. In *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops*, PERCOMW'07, pages 472–476, NY, USA.

Horn, P. (2001). Autonomic computing: Ibm's perspective on the state of information technology. *Computing Systems*, 15(Jan):1–40.

Huebscher, M. C. and McCann, J. A. (2008). A survey of autonomic computing degrees, models, and applications. *ACM Comput. Surv.*, 40(3):7:1–7:28.

Ibrahim, N., Le Mouel, F., and Frénot, S. (2009). Mysim: a spontaneous service integration middleware for pervasive environments. In *the international conference on Pervasive services*, ICPS' 09, pages 1–10, London, United Kingdom.

Kalasapur, S., Kumar, M., and Shirazi, B. A. (2007). Dynamic service composition in pervasive computing. *IEEE Transactions on Parallel and Distributed Systems*, 18:907–918.

Kiss, C. (2007). Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 2.0. W3C Working Draft 30 April 2007. http://www.w3.org/TR/2007/WD-CCPP-struct-vocab2-20070430/.

Klyne, G. (1999). Protocol-independent Content Negotiation Framework. RFC 2703.

Klyne, G., Reynolds, F., C.Woodrow, Ohto, H., Hjelm, J., Butler, M. H., and Tran, L. (2004). Composite capability/preference profiles (cc/pp): Structure and vocabularies 1.0. http://www.w3.org/TR/2004/REC-CCPP-struct-vocab-20040115/. W3C Recommendation.

Kongdenfha, W., Motahari-Nezhad, H. R., Benatallah, B., Casati, F., and Saint-Paul, R. (2009). Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *IEEE Transactions on Services Computing*, pages 94–107.

Kumar, M., Shirazi, B. A., Das, S. K., Sung, B. Y., Levine, D., and Singhal, M. (2003). Pico: A middleware framework for pervasive computing. *IEEE Pervasive Computing*, 2(3):72–79.

Li, X., Fan, Y., Madnick, S., and Sheng, Q. Z. (2010a). A pattern-based approach to protocol mediation for web services composition. *Information and Software Technology (IST)*, 52:304–323.

Li, Y., Lu, Y., Yin, Y., Deng, S., and Yin, J. (2010b). Towards qos-based dynamic reconfiguration of soa-based applications. In *APSCC*, pages 107–114.

McKinley, P. K., Sadjadi, S. M., Kasten, E. P., and Cheng, B. H. (2004). Composing adaptive software. *Journal of IEEE Computer*, 37:56–64.

Mélisson, R., Romero, D., Rouvoy, R., and Seinturier, L. (2010). Supporting Pervasive and Social Communications with FraSCAti. In *3rd DisCoTec Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services*, CAMPUS'10, Amsterdam, Netherlands.

Mukhtar, H., Belaïd, D., and Bernard, G. (2008a). A model for resource specification in mobile services. In *the 3rd international workshop on Services integration in pervasive environments*, SIPE '08, pages 37–42, Sorrento, Italy.

Mukhtar, H., Belaïd, D., and Bernard, G. (2008b). A policy-based approach for resource specification in small devices. In *Proceedings of the second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 239–244, Valencia, Spain.

Mukhtar, H., Belaïd, D., and Bernard, G. (2009). User preferences-based automatic device selection for multimedia user tasks in pervasive environments. In *the 5th International Conference on Networking and Services*, ICNS' 09, pages 43–48, Valencia, Spain.

Mukhtar, H., Belaïd, D., and Bernard, G. (2011). Dynamic user task composition based on user preferences. *ACM Transactions on Autonomous and Adaptive Systems*, 6(1):4:1–4:17.

Open Mobile Alliance (OMA) (2001). User agent profile (uaprof) specifications. http://www.openmobilealliance.org/.

Open SOA Collaboration (2005). Sca policy framework v1.00 specifications. http://www.osoa.org/.

Open SOA Collaboration (2007). SCA Assembly Model Specification V1.00. http://www.osoa.org/.

OSGI (1999). Open services gateway initiative. http://www.osgi.org.

Papazoglou, M. and van den Heuvel, W.-J. (2007). Service oriented architectures: Approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415.

Papazoglou, M. P. (2003). Service-oriented computing: concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12.

Passani, L. and Trasatti, A. Wurfl - wireless wireless universal resource file. Open source project. http://wurfl.sourceforge.net. Last Accessed on 20th August, 2009.

Pinto, M. and Fuentes, L. (2007). Ao-adl: an adl for describing aspect-oriented architectures. In *Proceedings of the 10th international conference on Early aspects*, pages 94–114. Springer-Verlag.

Ranganathan, A. and Campbell, R. H. (2004). Autonomic pervasive computing based on planning. In *1st International Conference on Autonomic Computing*, ICAC, pages 80–87, New York.

Román, M. and Campbell, R. H. (2003). A middleware-based application framework for active space applications. In *Proceedings of theInternational Conference on Middleware*, Middleware '03, pages 433–454, Rio de Janeiro, Brazil.

Román, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R. H., and Nahrstedt, K. (2002). Gaia: a middleware platform for active spaces. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):65–67.

Ruz, C., Baude, F., and Sauvan, B. (2010). Component-based generic approach for reconfigurable management of component-based soa applications. In *Proceedings of the 3rd International Workshop on Monitoring, Adaptation and Beyond*, MONA '10, pages 25–32, New York, NY, USA. ACM.

Ruz, C., Baude, F., and Sauvan, B. (2011). Flexible adaptation loop for component-based soa applications. In *7th International Conference on Autonomic and Autonomous Systems (ICAS 2011)*.

Schuhmann, S., Herrmann, K., and Rothermel, K. (2008). A framework for adapting the distribution of automatic application configuration. In *Proceedings of the 5th international conference on Pervasive services*, ICPS'08, pages 163–172, Sorrento, Italy.

Seinturier, L., Merle, P., Fournier, D., Dolet, N., Schiavoni, V., and Stefani, J.-B. (2009). Reconfigurable SCA applications with the frascati platform. In *Proceedings of IEEE International Conference on Services Computing*, SCC'09, pages 268–275, Bangalore, India.

Silva Filho, R. S. and Redmiles, D. F. (2005). Striving for versatility in publish/subscribe infrastructures. In *Proceedings of the 5th international workshop on Software engineering and middleware*, SEM '05, pages 17–24.

Sousa, J. P. and Garlan, D. (2002). Aura: An architectural framework for user mobility in ubiquitous computing environments. In *the 3rd Working IEEE/IFIP Conference on Software Architecture*, pages 29–43, Montréal, Canada.

Sousa, J. P., Poladian, V., Garlan, D., Schmerl, B., and Shaw, M. (2006). Task-based adaptation for ubiquitous computing. *IEEE Transactions on Systems, Man, and Cybernetics*, 36(3):328–340.

Spalazzese, R. and Inverardi, P. (2010). Mediating connector patterns for components interoperability. In *The fourth European Conference on Software Architecture*, ECSA'10, pages 335–343, Copenhagen, Denmark.

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, Boston, MA, USA, 2nd edition.

UPnP Forum (2008). UPnP Device Architecture 1.1. http://www.upnp.org.

Weiser, M. (1991). The computer for the twenty-first century. *Scientific American*, 9:94–100.

Zhai, Y., Zhang, J., and Lin, K.-J. (2009). Soa middleware support for service process reconfiguration with end-to-end qos constraints. In *IEEE International Conference on Web Services*, ICWS '09, pages 815–822, Washington, DC, USA. IEEE Computer Society.