Adapting Abstract Component Applications Using Adaptation Patterns

Imen Ben Lahmar*, Djamel Belaïd*, and Hamid Muktar†
*Institut Telecom; Telecom SudParis, CNRS UMR SAMOVAR, Evry, France
Email: {imen.ben_lahmar, djamel.belaid}@it-sudparis.eu
†National University of Sciences and Technology, Islamabad, Pakistan
Email: hamid.mukhtar@seecs.edu.pk

***Abstract*—Using a component-based approach, applications can be defined as an assembly of abstract components, requiring services from and providing services to each other. At the time of execution, they are mapped to the concrete level after identifying the deployed components. However, several problems can be detected at init time that prevent the mapping to be achieved successfully, e.g., heterogeneity of connection interfaces. Moreover, applications in pervasive environment are challenged by the dynamism of their execution environment due to, e.g., users and devices mobility, which make them subject to unforeseen failures.**

Both of these problems imply mismatches between abstract and concrete levels detected at init time or during the execution. Therefore, abstract applications have to be adapted to carry out their mapping and their execution.

In this article, we propose a new dynamic structural adaptation approach for abstract applications. Our approach is based on adaptation patterns that provide solutions to the captured mismatches between abstract and concrete levels. We also compare and contrast our approach with the existing ones concluding that our approach is not only generic, but it is also applicable both at init time and at runtime.

***Index Terms*—Adaptation patterns, adapter template, abstract application, mismatch, pervasive environments.**

I. Introduction

The recent research work related to automatic service composition in pervasive environments has gained much maturity, and together with advancements in various technologies, this concept has reached a level where the life of an ordinary user is completely automatized. Emphasis has been on the automatic selection of services for users, without their intrusion, in the pervasive environment. In most cases, such an approach considers an abstract user task on the user device, which leads to automatic selection of services across various devices in the environment, according to the current context.

For example, consider a video player application that provides the functionality of displaying video to the user. The user is also able to control the playback of the application. The application is represented by an assembly of abstract components, which describe only the services required or provided by the application namely, controlling, decoding and displaying video. The application has to be mapped to the concrete components to achieve its realization.

The complexities involved in designing and realizing such applications have been identified and addressed by many previous approaches [2] [5] [14]. Mostly, they consider both the functional and non-functional aspects of the application. For example, in one of our previous work, we have presented a mapping algorithm that maps an abstract application to concrete one considering functional aspects of the application —interfaces and message interactions— as well as non-functional aspects like user preferences, devices' capabilities and network protocol heterogeneity to ensure the mapping of services to the components [14].

While the existing approaches may assume that a mapping from abstract to concrete application can be done effortlessly, many problems may rise that prevent it to be achieved successfully. As a first case, consider the heterogeneity of interfaces of different components or devices, heterogeneity of interaction messages, etc. Furthermore, applications 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. An automatic adaptation is to remap the abstract description of application to the concrete level. The remapping corresponds to the reselection of new concrete components to replace some others.

Both of these problems imply mismatches between abstract and concrete levels that may occur either at the time of mapping during initialization or even after the application has been executed. Thus, adapting abstract applications represents a crucial need to be considered in order to ensure their mapping to the concrete level and their execution.

The problem of adapting component-based models has been extensively studied in different contexts, notably in component-based applications. In the literature, we distinguish many relevant adaptive approaches that propose parametric or compositional mechanisms to adapt applications in pervasive environment (e.g., [2] [18] [19]). Parameterization techniques aim at adjusting internal or global parameters in order to respond to changes in the environment. Compositional adaptation is classified into structural and behavioural adaptations [13]. We mean by behavioural adaptation the modification of the functional behaviour of 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 article, we propose a dynamic structural adaptation approach based on adaptation patterns to provide solutions to the detected mismatches between abstract application and the concrete level. Adaptation patterns are injected into the abstract application to provide extra-functional services allowing its mapping and its execution. Thus, the abstract application is transformed to another one that ensures its mapping and its execution. To facilitate the description of the adaptation patterns, we define a generic adapter template that encapsulates the main features of an adapter.

We are not interested in describing the detection of adaptation context; rather, our objectives are: 1) to define an adapter template and to give examples of adaptation patterns based on this template and 2) to show how adaptation patterns are used to adapt the abstract application and hence the concrete one.

The remainder of this paper is organized as follows. Section II mentions and classifies examples of factors triggering the adaptation of abstract application. Section III describes the principle of our structural adaptation approach illustrated by examples of adaptation patterns. In Section IV, we present an example scenario through which we show how patterns are used to adapt dynamically an application. In Section V, we present some implementation details. Section VI provides an overview of existing related approaches as well as their limitations. Finally, Section VII concludes the article with an overview of our future work.

II. Adaptation Contexts

A generalized notion of context has been proposed in [1] 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 application. We are interested in contexts that represent the mismatches between abstract and concrete levels. 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.

We classify the factors triggering the adaptation of the abstract description into three categories: 1) factors related to the software characteristics of devices, 2) factors associated with the network characteristics of devices and 3) factors related to the hardware characteristics of devices. Our intent through the given classification is to cover the different aspects related to software, network or hardware sides that may trigger the adaptation of the abstract application.

The software factors are related to the software components of the application through which it describe its functionalities. A mismatch related to software factors includes interface mismatches due to different syntactic representation, differences of the interaction messages supported by components, requiring component wrappers to hide implementation heterogeneity, or adding proxy components to control access, etc.

In hardware factors, we consider the factors associated with the hardware characteristics of devices. A change in hardware configuration may lead to change in the context, requiring adaptation. Such changes may imply improvement or decline in the device capacities, e.g., high CPU usage, reduced capacity of memory storage, attachment of new hardware components, etc.

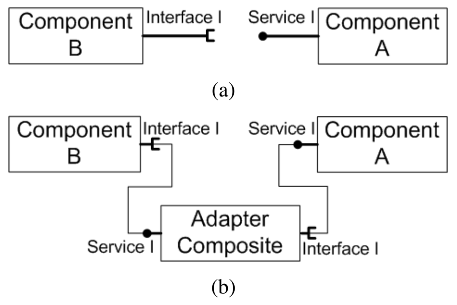

Fig. 1. Transforming abstract application using Adapter composite

In addition to the software and hardware factors, there are several challenges related to the network characteristics of a device that may require the adaptation of the application, e.g., use of different interfaces of network connections such as changing to Bluetooth from Wi-Fi if the latter disappears, the fluctuation in the quality of network signals, etc.

The example factors are some of recurrent causes that are responsible for the mismatches between abstract and concrete levels. There may be other factors that trigger the adaptation of the abstract description of the application to support the changes of the environment (e.g., user preferences); however, we do not deal with them in this article.

In the next section, we detail the proposed adaptation approach to overcome the detected mismatches that are triggered by some factors related to software, hardware or network characteristics of devices.

III. Adaptation Patterns

A. Principle of our Approach

To overcome the captured mismatches between abstract and concrete levels, we propose to transform an abstract application to another one that ensures its mapping and its execution. The transformation consists of injecting extra-functional adapters into the abstract application.

As shown in figure 1, an adapter composite is injected to adapt the interaction 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. Thus, the abstract application is transformed by adding extra-functional behaviour to achieve its mapping or its execution.

We note that adapters are identified at execution time without the assistance of the designer. This can be done by using a set of adaptation policies defined at the design time. These policies can then relate the adaptation pattern with the execution context. However, in the present work, we do not tackle the identification of the used adapters that remains an objective for the future work.

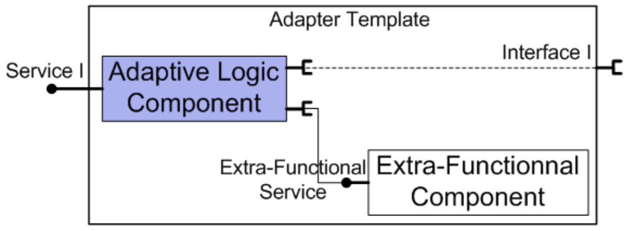

Fig. 2. Generic adapter template

B. Adapter Template

As the basis for our approach we propose to use adaptation patterns as adapter composites to provide solutions for the detected mismatches between abstract and concrete levels. Patterns have component-based descriptions that encapsulate extra-functional services for the adaptation of abstract applications.

We have defined an adapter template to be used for the description of adaptation patterns. Figure 2 shows the description of the adapter template, which consists of an ”adaptive logic” component and an ”extra-functional” one.

The extra-functional component provides transformation services allowing, e.g., encryption, compression, etc. The adaptation pattern provides an abstract description for the extra-functional component to be mapped following the matching algorithm [14]. Thus, the extra-functional component is identified dynamically without the assistance of designer. Its offered service will be used by an adaptive logic component which acts as an intermediate between the abstract and the extra-functional components.

For this, the adaptive logic component is considered as the main element of the adapter template. And each adaptation pattern should contain at least the adaptive logic component, if it does not require any transformation service to overcome the mismatch (see Section IV). The implementation of the adaptive logic component is generated, since it depends to the business interfaces of abstract components. As shown in figure 2, the adaptive logic component requires the offered service of the extra-functional component and a service implementing the interface I, which corresponds to the service I of the component A. Otherwise, the component provides a service implementing the interface I as the required service of the component B.

Using this specific structure of the adapter template has an advantage, on the one hand, to separate the adaptive logic from the functional logic of the application. Thus, it is possible to modify the adaptive logic with respect to the components' descriptions. On the other hand, it allows providing an abstract description for the adaptation actions, corresponding to extra-functional components, to be mapped following the matching algorithm. Thereby, the separation between adaptive logic and extra-functional components facilitates the generation of the adaptive logic that plays an intermediary role between abstract components and extra-functional ones.

Due to space limitation, in this section, we only discuss three adaptation patterns that are defined based on our introduced adapter template.

C. Example Adapter Patterns

1) Compressor Pattern: A compressor pattern is used to handle a mismatch between an abstract application and a concrete level triggered by a network factor, which is the fluctuation of the network signal used by devices. Thus, if the signal strength is high, bigger data can be sent over network. However, if the signal strength is weak; data should be compressed for a quick transfer.

The compressor pattern has the given description by figure 3. It consists of an adaptive compression component and a compressor one. The abstract compressor component will be mapped to the concrete level to identify the corresponding concrete component that implements the described compression interface. However, the adaptive compression component will be generated to implement the interface I as required by the component B. Its implementation contains an invocation of the provided service of the compressor component in addition to the service I.

2) Decompressor Pattern: A decompressor pattern is used whenever a compressor pattern is handled by a device to overcome the weakness of the network signal. Indeed, within a compressor pattern, we require a decompressor pattern to decompress the data before using it by the component A.

The decompressor pattern is described following the adapter template as shown in figure 3. It contains an adaptive decompression component and a decompressor one. While the decompressor component is mapped to the concrete level, the adaptive decompression component is generated to provide a service I. The implementation of the adaptive logic component consists on applying a decompression algorithm supported by the service of the decompressor component before invoking the service I provided by the component A.

3) Proxy Pattern: The proxy pattern, as defined by Gamma et al. [9], provides solutions to problems related to inaccessible software entities. Thus, it would be useful to overcome the network factor related to different interfaces of connections as detailed in the next section IV.

Figure 4 gives a component-based 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. It contains only a proxy component representing the adaptive logic component that forwards the call of the service I to the component A.

IV. Example Scenario Using Adaptation Patterns

Referring back to the video player application described in the introductory section, figure 5 shows an abstract description of the Video player application that consists of three components: a *VideoDecoder* component, a *DisplayVideo* component and a *Controller* Component. The *Controller* component sends a command to the *VideoDecoder* component to decode a stored video. The *VideoDecoder* component decodes a video into appropriate format. Once the video is decoded, it is passed

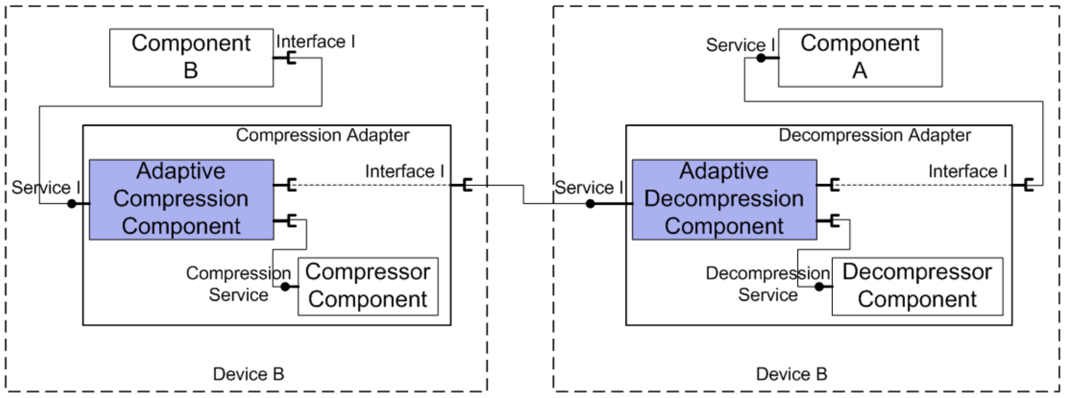

Fig. 3. Compressor and decompressor adaptation patterns

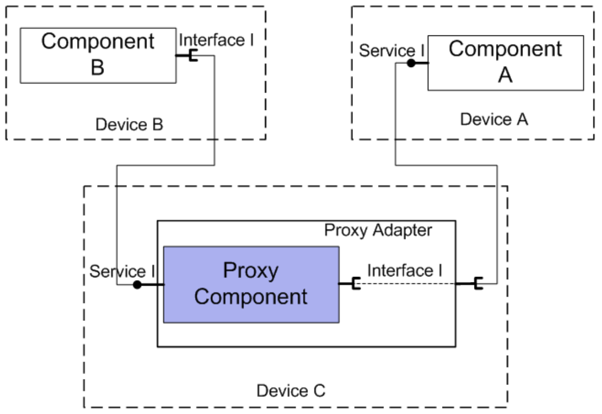

Fig. 4. Proxy adaptation pattern

to the *DisplayVideo* component to play it. This is done using the service provided by the *DisplayVideo* component through an appropriate programming interface.

The given execution environment consists of a Smartphone (SP) device that supports a Wi-Fi interface connection and contains the *Controller* and the *VideoDecoder* components. There is also a flat-screen (FS) device that uses a Bluetooth interface connection and maintains the *DisplayVideo* component. A laptop (LP) device is available in the execution environment and supports both interfaces of connection (i.e., Bluetooth and Wi-Fi). However, LP does not provide any services described in the video player application. Thus, only the smartphone and the screen devices would be used to support the application mapping.

However, both of these devices cannot interact between them, since they support different connection interfaces. Thus, there is a mismatch between the given abstract description of the video player application and the concrete level because of a network factor that causes the failure of the mapping. Therefore, the Video Player application should be adapted to achieve its mapping.

To overcome the heterogeneity of connection interfaces, we propose to use a proxy adaptation pattern to create a representative for the *DisplayVideo* component. As described in III-C3, the composite of the proxy pattern contains only a *DisplayVideoProxy* component representing the adaptive logic. It requires the *DisplayVideoService* to forward the call of the *VideoDecoder* component to the *DisplayVideo* component. The proxy pattern is generated on the laptop device since this latter supports Wi-Fi and Bluetooth connections. As a result, the application is transformed, as shown in figure 6, to contain the DisplayVideoProxy component in addition to its own components.

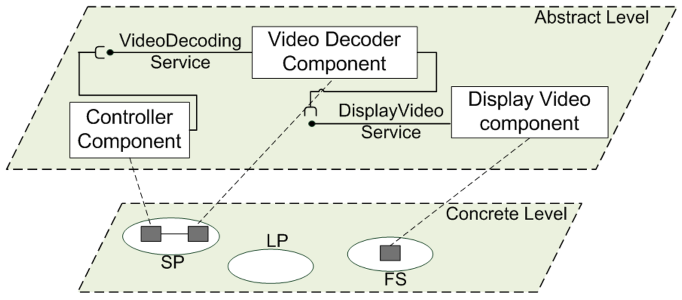

Fig. 5. Video Player application

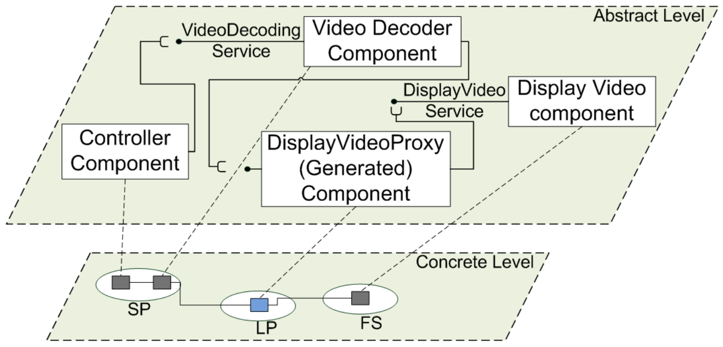

Fig. 6. Adaptation of the Video Player application

V. Implementation

We have implemented, as a proof of concept, a java prototype of a service allowing the automatic generation of pattern's components following the description of adapter template. SCA (Service Component Architecture) [15] is used to describe components since it provides the ability to write applications abstractly and then map them to the concrete components. One of the main features of the SCA component model is that it is independent of particular technology, protocol, and implementation.

The open source Javassist [1] (JAVA programming ASSISTant) library is used to generate the byte code of an adaptive logic component that implements a specified interface I. For this purpose, we use the `java.lang.reflect` package to obtain reflective information about methods of the interface I (i.e., names, parameters, etc).

Similarly, if the adaptive logic component requires an extra-functional component, the interface of this latter is introspected to be used for the implementation of the adaptive logic component.

In case of our example Video Player application, the interconnectivity between devices is represented by a graph [14]. For the given scenario, there will be no connection between smartphone and flat-screen devices in the graph, representing an interface mismatch. However, connections between smartphone and laptop, and between laptop and flat-screen are detected in the graph, which indicates a proxy pattern should be generated in the laptop device to overcome interface mismatching. The pattern consists of the DisplayVideoProxy component that implements the introspected interface of the DisplayVideo service and whose byte code is generated dynamically using the Javassit library.

VI. Related Work

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

In [17], Sadjadi et al. propose a general programming model called transparent shaping that supports the design and the development of adaptable programs from existing programs. In transparent shaping, an application is augmented with hooks that intercept and redirect interaction to adaptive code. Integration of the adaptive code is a two-phase process, in the first phase; the application is transformed into a new adapt-ready application by weaving hooks into the application code. In the second phase, usually executed at run-time, an adaptation infrastructure is inserted dynamically using these hooks.

Compared to our approach, the major drawback of their approach is that the identification of the hooks is done at development time. Besides, the program should be in ready state, using a static transformation, to support the insertion and the removal of adaptive code at runtime. However, in our case, we provide a dynamic adaptation for applications at init time as well as at runtime without any transformation in advance.

[16][8] present an AO-ADL (Aspect Oriented-Architecture Description Language) language. AO-ADL considers that components model either crosscutting (named aspectual component) or non-crosscutting behaviour (named base component) exhibiting a symmetric decomposition model. Towards this challenge, they propose to use aspectual connectors that provide support to describe and to weave aspectual components to the regular ones. These connectors are described following a template connector.

However, the instantiation of the connector template is done at design time, which limits the possibility to extend applications with new aspects. Furthermore, they consider connector template per aspect. However, it is not possible to define as many connector templates as possible aspects.

In another related work, AO-OpenCom [10] [18] that is an Aspect Oriented Middleware platform, provides a dynamic reconfiguration of distributed aspects. The platform builds on the OpenCom component model [7] and the distributed component framework. Aspect composition in AO-OpenCom employs components to play the role of aspects. Aspects are composed at runtime using so-called interceptor-connectors that support the dynamic insertion of aspect components. Aspects can be added to a system at run-time using an aspect-oriented MOP that allows a fine-grained introspection and an adaptation of cross-cutting behaviour.

While their approach is quite general, we can identify two limitations compared to our proposed approach. First, aspects are used only to adapt OpenCom-based application at runtime. Second, the join points are already defined by designer, which limits the number of adaptations.

[19] presents a WComp middleware that uses the concept assembly of aspects (AA) to modify the internal component assembly of event-based web services [11]. Aspects of assembly are pieces of information describing how an assembly of components will be structurally modified. In such mechanisms, aspects are selected either by the user or by a self-adaptive process and composed by a weaver with logical merging rules. The result of the weaver is then projected in terms of pure elementary modifications of components assemblies.

However, the weaving and the validation rules are predefined by the developer at design time, which limits the number of adaptation. Moreover, they do not take into account to the adaptation of applications at initialization.

In [3], Becker et al. propose an adaptation model which is built upon a classification of component interfaces' mismatches. To cover these mismatches, they identify a number of patterns to be used for eliminating the interfaces' mismatches. They classify the adaptability pattern into two categories; functional adaptation pattern to bridge the functional component incompatibilities and extra functional adaptation pattern to increase a single or several quality attributes of the component being adapted.

However, these adaptation patterns are identified at design time to achieve the adaptation of applications at runtime. Besides, they limit the adaptation contexts to the mismatches between components' interfaces. Thus, they do not take into account the network and hardware factors related to characteristics of devices.

Other related work in this area [12] [6] have also investigated matching of Web service interfaces by providing a classification of common mismatches between service interfaces

[1] http://www.csg.is.titech.ac.jp/~chiba/javassist/

and business protocols, and introducing mismatch patterns. These patterns are used to formalize the recurring problems related to the interactions between services. The mismatch patterns include a template of adaptation logic that resolves the detected mismatch. Developers can instantiate the proposed template to develop adapters. For this purpose, they have to specify the different transformation functions.

We can identify two important limitations compared to our approach. First, the mismatch patterns are limited to the interfaces and protocols mismatches. Thus, they do not consider the other different software, network and hardware factors. Second, the specification of adapters is done with the help of developers. However, in our approach, we are able to specify dynamically the different components of a used pattern; by generating the implementation of its adaptive logic component and mapping the extra-functional one following our matching algorithm [14].

VII. Conclusion and Future work

In this article, we have proposed an approach for dynamic structural adaptation of abstract applications. We limit the adaptation contexts on factors implying mismatches between abstract application and the concrete level. These factors are related to software, hardware and network characteristics of devices and they are detected at init time or during the execution of the application.

To overcome these mismatches, we propose to use adaptation patterns that provide extra-functional behaviour to the abstract applications. The adaptation patterns are described following our generic adapter template, which consists of adaptive logic and extra-functional components. While the adaptive logic component has a generated implementation, the extra-functional component is identified following a mapping process in [14].

Using this approach allows to separate the extra-functional logic from the business one, and hence, to add or remove adaptation patterns dynamically from the abstract application whenever there is a need.

We are looking forward to integrate the implemented prototype in our SCA platform [4] and the Newton SCA runtime[2]. And in the near future, we will tackle the identification of adaptation patterns that are used to support the adaptation of abstract applications.

VIII. ACKNOWLEDGMENTS

This work is partially supported by French ANR through Seamless (SEamless and Adaptive Services over MultipLe AccEsS NetworkS) project number 07TCOM018.

References

[1] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307, 1999.

[2] Christian Becker, Marcus Handte, Gregor Schiele, and Kurt Rothermel. Pcom - a component system for pervasive computing. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, page 67, Washington, DC, USA, 2004. IEEE Computer Society.

[3] Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky, and Massimo Tivoli. Towards an engineering approach to component adaptation. In *Architecting Systems with Trustworthy Components*, pages 193–215, 2004.

[4] Djamel Belaïd, Hamid Mukhtar, Alain Ozanne, and Samir Tata. Dynamic component selection for sca applications. In *I3E*, pages 272–286, 2009.

[5] Sonia Ben Mokhtar, Nikolaos Georgantas, and Valérie Issarny. Cocoa: Conversation-based service composition in pervasive computing environments with qos support. *J. Syst. Softw.*, pages 1941–1955, 2007.

[6] Boualem Benatallah, Fabio Casati, Daniela Grigori, H. R. Motahari Nezhad, and Farouk Toumani. 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, 2005.

[7] Geoff Coulson, Gordon Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama. A component model for building systems software. In *IASTED Software Engineering and Applications (SEA04)*, 2004.

[8] Lidia Fuentes, Nadia Gámez, Mónica Pinto, and Juan A. Valenzuela. Using connectors to model crosscutting influences in software architectures. In *ECSA*, pages 292–295, 2007.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] Paul Grace, Bert Lagaisse, Eddy Truyen, and Wouter Joosen. A reflective framework for fine-grained adaptation of aspect-oriented compositions. In *SC'08: Proceedings of the 7th international conference on Software composition*, pages 215–230. Springer-Verlag, 2008.

[11] Vincent Hourdin, Jean-Yves Tigli, Stéphane Lavirotte, Gaëtan Rey, and Michel Riveill. Slca, composite services for ubiquitous computing. In *Mobility '08: Proceedings of the International Conference on Mobile Technology, Applications, and Systems*, pages 1–8. ACM, 2008.

[12] Woralak Kongdenfha, Hamid Reza Motahari-Nezhad, Boualem Benatallah, Fabio Casati, and Regis Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *IEEE Transactions on Services Computing*, pages 94–107, 2009.

[13] Philip. K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A taxonomy of compositional adaptation. Technical report, 2004.

[14] Hamid Mukhtar, Djamel Belaïd, and Guy Bernard. A graph-based approach for ad hoc task composition considering user preferences and device capabilities. In *Workshop on Service Discovery and Composition in Ubiquitous and Pervasive Environments*, New Orleans, LA, USA, dec 2007.

[15] Open SOA Collaboration. Service component architecture (sca): Sca assembly model v1.00 specifications. http://www.osoa.org/, 2007.

[16] Mónica Pinto and Lidia Fuentes. Ao-adl: an adl for describing aspect-oriented architectures. In *Proceedings of the 10th international conference on Early aspects*, pages 94–114. Springer-Verlag, 2007.

[17] S. Masoud Sadjadi, Philip K. McKinley, and Betty H. C. Cheng. Transparent shaping of existing software to support pervasive and autonomic computing. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7. ACM, 2005.

[18] Bholanathsingh Surajbali, Geoff Coulson, Phil Greenwood, and Paul Grace. Augmenting reflective middleware with an aspect orientation support layer. In *ARM '07: Proceedings of the 6th international workshop on Adaptive and reflective middleware*, pages 1–6. ACM, 2007.

[19] Jean-Yves Tigli, Stephane Lavirotte, Gaëtan Rey, Vincent Hourdin, Daniel Cheung-Foo-Wo, Eric Callegari, and Michel Riveill. Wcomp middleware for ubiquitous computing: Aspects and composite event-based web services. *Annales des Télécommunications*, pages 197–214, 2009.

[2]http://newton.codecauldron.org/site/concept/ComponentModel.html