Mohammad Ullah Khan

Unanticipated Dynamic Adaptation of Mobile Applications

kassel
university
press

This work has been accepted by the faculty of Electrical Engineering and Computer Science of the University of Kassel as a thesis for acquiring the academic degree of Doktor der Ingenieurwissenschaften (Dr.-Ing.).

Advisers:
    Prof. Dr. Kurt Geihs
    Prof. Dr. Peter Herrmann

Additional Doctoral Committee Members:
    Prof. Dr. Claudia Leopold
    Prof. Dr. Jan Marco Leimeister

Defense day: 31st March 2010

Bibliographic information published by Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data is available in the Internet at http://dnb.d-nb.de.

Zugl.: Kassel, Univ., Diss. 2010
ISBN print: 978-3-89958-918-4
ISBN online: 978-3-89958-919-1
URN: http://nbn-resolving.de/urn:nbn:de:0002-9193

Printed in Germany

To my grandma

Mojirunnesa

I am who I am because of you

Abstract

Driven by the emergence of mobile and ubiquitous computing there is a growing demand for context-aware applications that can dynamically adapt to their runtime environment. Middleware support for providing such adaptation to mobile applications has been an attractive research and development issue for several years now. However, one of the major challenges to support adaptation is that they can not be always foreseen during the design time. In a ubiquitous computing environment, a number of mobile devices running adaptive applications supported by the instances of the same middleware may come and go in a particular adaptation domain in an unanticipated manner. Moreover, third party services may also appear and disappear with respect to that particular adaptation domain. In component-based application architectures, with the enhancement of integrating services, mobile applications may benefit from using the components that are provided by other users as well as using third party services. In an ideal case, for a user it should be sufficient to specify what he expects his application to do. At runtime the application will be composed of the available components and services within the adaptation domain. In practice, the core components may be provided by the application developer himself to ensure at least a minimum configuration of the application, while discovered components and services add flexibility of integrating new functionalities along with improving the quality of service.

The basis of the work is the results of the research projects MADAM [1] and its successor, MUSIC [2] that I have been involved in. However, none of these projects explicitly addresses the unanticipated adaptation. In this thesis I will present an approach to handle such unanticipated adaptations through adopting and extending the support provided by those projects. I have developed both a conceptual solution, consisting of the approach to address the unanticipated adaptation along with a brief description of the middleware and guidelines for the application developers. I have also implemented an initial version of the middleware based on these concepts. The middleware is tested as a proof of the main concepts, promised to be provided by this work. In order to present a complete picture of the solution MUSIC results are also briefly introduced. Furthermore, I specify and clearly identify where and how that results are updated in this work.

Zusammenfassung

Durch die Entstehung von "Mobile and Ubiquitous Computing" besteht eine wachsende Nachfrage nach kontextbewussten Anwendungen, die sich dynamisch während ihrer Rechenzeit an die Umgebung anpassen können. Middleware-Unterstützung für eine solche Adaption der mobilen Anwendungen ist seit mehreren Jahren ein attraktives Themengebiet der Forschung und Entwicklung. Eine der größten Herausforderungen zur Unterstützung dieser Anpassungsfähigkeit ist, dass diese nicht immer während des Entwicklungszeitraums vorhersehbar ist. In einem ubiquitären Umfeld können zahlreiche mobile Geräte, auf denen Applikationen ausgeführt werden, die durch Instanzen der gleichen Middleware unterstützt werden, unvorhersehbar innerhalb einer bestimmten Anpassungsdomain auftauchen und verschwinden. Darüber hinaus können Services von Dritten auch innerhalb dieser Anpassungsdomain auftauchen und verschwinden. In einer komponentenbasierten Anwendungsarchitektur mit der Möglichkeit Services zu integrieren können mobile Anwendungen Vorteile aus der gemeinsamen Nutzung von Komponenten anderer Benutzer sowie Services von Dritten ziehen. Im Idealfall sollte es ausreichen, dass ein Benutzer seine Anforderungen an seine Anwendung genau spezifiziert. Die Anwendung wird dann während der Laufzeit basierend auf den verfügbaren Komponenten und Services zusammengesetzt. In der Praxis sollten die Kernkomponenten jedoch vom Anwendungsentwickler selber bereitgestellt werden, um eine minimale ausführbare Konfiguration der Anwendung sicherzustellen, während die entdeckten Geräte und Services die Möglichkeit zur Integration neuer Funktionalitäten sowie zur Verbesserung der Servicequalität erlauben.

Diese Arbeit basiert auf den Ergebnissen des Projekts MUSIC [2] und dessen Vorgänger MADAM [1], in denen ich gearbeitet habe. Allerdings adressiert MUSIC nicht explizit unvorhersehbare Adaption. In dieser Doktorarbeit stelle ich einen Ansatz zur Behandlung solcher unvorhersehbaren Anpassungen vor, wobei ich auf den bisherigen Ergebnissen von MUSIC aufbaue und diese erweitere. Dazu entwickelte ich eine konzeptuelle Lösung bestehend aus dem eigentlichen Ansatz zur Behandlung unvorhersehbarer Adaptionen sowie einer kurzen Beschreibung der Middleware und Richtlinien für die Anwendungsentwickler. Weiterhin habe ich eine erste Version der Middleware basierend auf diesen Konzepten entwickelt und diese bezüglich der im Konzept beschriebenen Anforderungen getestet. Um ein möglichst vollständiges Bild der Lösung darzustellen, werden in dieser Arbeit MUSIC-Ergebnisse eingeführt und detailliert beschrieben wie diese durch mich erweitert wurden.

Acknowledgements

The first person I would like to thank is my doctoral advisor *Prof. Kurt Geihs*. It has been a pleasure to work in a group led by such a smart person. He has always been available for technical as well as non-technical discussions in order to proceed forward with the work. I am extremely grateful also to *Prof. Peter Herrmann*, who has supervised the thesis within a very strict time constraint.

Blessed with the proficient leadership of *Kurt*, the distributed systems research group has provided me a great working environment. It has been an excellent experience to work in the MUSIC team, also consisting of *Roland Reichle* and *Michael Wagner*, who have been always dedicated to the 'team work' with their intelligence and sincerity. *Christoph Evers* has also joined the MUSIC team at a later stage and he is as dedicated as the others. Besides the MUSIC team, I have shared my research ideas with *Steffen Bleul*, *Thomas Weise*, *Michael Zapf*, *Diana Comes* and *Hendrik Skubch* and received constructive feedback. *Philipp Baer* worked in a different field; but his dedication to his work and hobbies have inspired me for going forward. Being a Linux expert, he has always been there with his helping hand. Last but not the least, *Iris Rossbach* and *Thomas Kleppe* have helped maintaining a healthy infrastructure in the group.

Besides my colleagues at the University of Kassel, I have come across a good number of people, who have been a part of the MADAM and the MUSIC projects. I have worked closely with many of them on different research topics within the project. *Eduardo Soldana* and *Jorge Lorenzo* from Integrasys, with their expertise in Mobile applications, have provided constructive suggestions to get my middleware running on real mobile devices. *Erlend Stav* from SINTEF has shared his experiences and opinions on many adaptation-related concepts over the last four and a half years. *Nearchos Paspallis* from the University of Cyprus has been quite innovative with his ideas, especially in the topic of Context awareness. It has been a great experience for me working in projects, where *Svein Hallsteinsen*, *Geir Horn* and *Jacqueline Floch* have managed both technical and social aspects very efficiently. However, the list of innovative and efficient team players in those projects will only grow. I am very thankful to all of them for all their great support and team work.

Living in Kassel has offered me the luxury of getting in touch of a number of nice friends. *Ovi*, *Shohag*, *Mehbub*, *Nasir*, *Sumans*, *Shantoo*, *Mamun*, *Pavan*, *Prasant*, *Srinidhi, Imran*, *Ratul* all have been a part of my social life here. They have been there in the cricket ground as well as in the gatherings during the weekend. Besides being a social part of my Kassel life, *Himu* has contributed to the insightful discussion of the scenario for my thesis.

I am grateful to my parents and grandma, who have always inspired me over the telephone, while I have been ten thousand kilometers away from home. I am thankful to my wife *Aklima Nahar*, for I had to spend so much time on working even during the weekend, while she was expecting me to share the time with her. She had to sacrifice a lot of her travelling plans around the Europe only because I was busy.

Table of Contents

List of Figures

List of Tables

Part I Background

1 Introduction

Mobile and ubiquitous computing introduce a growing demand for applications that are able to adapt to the dynamically changing environment, its resources and user preferences as well as to the availability or disappearance of devices and services. Applications running in such environment are characterized by its distributed nature, for example, all components of a component-based application do not necessarily have to be provided by a single user or a single device, rather they may be offered by different devices and used in a transparent way. Moreover, the devices may be hosting components from independent developers and therefore, their availability and usability cannot be always foreseen. Moreover, in a ubiquitous computing environment, third party services may provide the functionalities, offered by a particular component, and may replace the component in the application configuration. The usage of alternating component realizations, services etc. creates different variants of the application through composition [3] of a new set of components and/or instantiation of particular components with a new set of properties which may depend on particular parameters [4]. In such cases, adaptations are supported by choosing from different variants of the application, comprising components and services that provide the same set of functionalities with a changed quality of service.

The concept of adaptation and context awareness of ubiquitous and mobile applications has been subject to research for several years. With the widespread use of such applications the need for the adaptation of applications, in order to be benefitted through using services and devices appearing and disappearing in an unanticipated manner during the runtime, is becoming an issue of immense interest to the research community as well as to the application developers. In the literature, the term 'unanticipated' has been used with different meanings because all adaptations must remain unanticipated until some points [5]. A popular understanding of the term 'unanticipated' is described as '*which has not been foreseen at design time*' [6] [7]. Therefore, the unanticipated software adaptation can also be understood to mean the software adaptations that are not anticipated until the execution of that software is started [8].

Manuel Oriol addresses the software evolution problem in his thesis [40], where he distinguishes between static and dynamic software evolution, identifying the time of changes. Based on the anticipation of such changes, he defines unanticipated evolution as consisting in '*evolution that has not been foreseen by the programmer'*. The complexity and the challenges of providing adaptation solutions depend on the time of anticipating the adaptation. The adaptations foreseen during the design time are the easiest to achieve. The challenge increases for the adaptations that are to be handled during the deployment of the application. This becomes even more difficult while the application is running.

For applications realized by components and services, the main tasks involved are keeping track of the runtime availability of the components and the services, calculating the possible variants of the application when alternating realizations are possible, choosing the best fitting variant among them based on some qualitative measurements, and then instantiating the components, creating service proxies and connecting the components and the services to configure the chosen variant of the application. In order to present a meaningful and seamless adaptation to the user, such tasks need to be performed without interrupting the application and in an efficient manner. This imposes a big challenge, especially for resource-limited mobile devices, when the number of possible configurations becomes quite large. Measuring the appropriate fitness value is also a big challenge. For example, in the utility function-based approach [9], the fitness of a particular application variant to a certain context situation is calculated by evaluating its utility function. The utility of an application variant is influenced by the context and resource dependencies, which varies from component to component. Moreover, in the case of the unanticipated adaptation, the number of variants may not be estimated beforehand and therefore, the adaptation reasoning approach needs to ensure a reasonable adaptation in quick time; e.g., in a few seconds.

In an aim to support unanticipated adaptation of mobile applications, in the following subsections, I provide more insight into the problem defining related terms and clarifying them with the help of an appropriate scenario.

1.1 Adaptation in Terms of Anticipation

The developer of an adaptive application can foresee some of the adaptation behaviors, while some other behaviors can not be foreseen while developing the application; rather they must be handled when the application is running. This inspires the need for a middleware platform that supports such unforeseen adaptation. In the case of ubiquitous computing, especially for the component-based applications supported by a particular middleware platform, components realizing the application may be provided by numerous developers so that a particular application may benefit from others' development in the aim to provide new functionalities with a better quality of service to its user. We can think of the realistic situation that the developers of such applications may or may not be aware of each others' development. Therefore, a particular developer can not always foresee the applications and components developed by others and eventually their usage in improving the quality of service of his application. There is one more case, where the application may also benefit from third party services, which do not target a particular platform; but can be discovered in the ubiquitous computing environment and used for different applications in general. A particular developer may not foresee the availability of such services; but he can express the need and be aware of the possible usage scenario of such services.

Based on the above discussion, I provide a definition of the adaptation problem in terms of its anticipation aspect from the view point of a particular application developer.

1.1.1 Anticipated Adaptation

> *"Anticipated adaptation is defined as an adaptation behavior, which is foreseen by a developer during the development of an application."*

The actual adaptation may still take place at runtime; but the developer has an insight of how the application will be adapted or realized. The same is true for developers, who are developing software components for particular applications. Even though an individual component developer may not be the same person as the application developer; but he must have knowledge of the target application that his components will be used for.

1.1.2 Semi-anticipated Adaptation

> *"Semi-anticipated adaptation is defined as an adaptation behavior, which can be partially foreseen by the developer during the application development process."*

For such adaptations, we particularly refer to the usage of third party services in realizing application functionalities. The reason behind such consideration is that an application developer may foresee his application using some third party services; but he is not aware of the realization of his application during the development process.

1.1.3 Unanticipated Adaptation

In connection with the component-based development of adaptive applications,

> *"Unanticipated adaptation is defined as an adaptation behavior that incorporates components from possibly a number of different developers, who have no prior knowledge of each other's development, to realize the application at runtime."*

Such adaptation simplifies the development process because one particular developer may independently focus only on what he is developing (as long as it is compliant to the platform). It is left to the middleware platform to integrate the components to realize the application.

From a user's point of view, such adaptations add extra flexibilities so that while he selects to run the application, he may express only what his application is supposed to do, while the actual realization of the application may involve components (and of course, services) provided by other users in the ubiquitous computing environment. At runtime, he may also vary the set of functionalities by adding new functionalities, removing existing functionalities or choosing a different set of functionalities that he expects his application to provide, without bothering to know how those will be realized[1].

One notable difference between the anticipated and the unanticipated adaptation is that in the case of the anticipated adaptation individual developers have prior knowledge of each others' development and therefore, one may foresee the realization possibility of his application using components from others. However, in the case of unanticipated adaptation, the developers are completely independent of each other and therefore, one does not need to know or even foresee about others' development.

In this work, I aim at providing a solution in terms of concepts, design and implementation of a middleware platform that supports such unanticipated adaptations.

1.2 Motivating Scenario

In order to provide a thorough understanding of the unanticipated adaptation, making clear distinctions between the meanings of the terms 'anticipation' and 'unanticipation', I provide a scenario, where a mobile user is supported by an unanticipated adaptive

[1] For practical applications such flexibilities are only meaningful, when large number of interacting users is available, which I vision as a feature of the future ubiquitous computing environment.

mobile application assisting him during his tours. Afterwards, the scenario is analyzed to identify the adaptations and the level of unanticipation.

Keukenhof in Lisse is the world's biggest tulip garden, which is open for public in the spring for two months between March and May. Thomas, a student of the University of Kassel, plans to visit it during a weekend. He has a car equipped with a computer supporting the navigation. He has a mobile device using the U-MUSIC[2] middleware and running a U-MUSIC adaptive application named UnanticipatedTravelAssistant, which also provides him navigation support along with creating itinerary, processing images and video, viewing maps etc. The application supports updating its functionalities as per the users' need and improving the quality of service by plugging in new services and components discovered at runtime.

1.2.1 Scene 1 - Traveling to the Netherlands

Marc, a friend of Thomas will also join him from Duisburg. Therefore, Thomas first plans the route to Duisburg from Kassel with his mobile device and gets in the car. While in the car, the presence of the car computer is detected and its screen is used instead of the less convenient screen of the mobile device to show the navigation information. Also, the car provides a navigation system with a better quality and therefore, the application in the mobile device automatically uses this navigation facility instead of the one in the mobile device. This also saves the battery power of the mobile device, while the stored information like the destination address or Thomas's user profile can be used from the mobile device. Thomas starts driving towards Duisburg. The car computer also supports a Head-up display (HUD), while the mobile device has the text-based user interface as well as the voice-based hands-free user interface. During the driving the application is automatically configured to use voice commands. At the start, the streets are not busy because it is early in the morning and therefore, the screen of the car computer is used to show navigation information. However, after driving for about one and a half hours, the traffic increases greatly and the relatively busy streets contribute to the automatic selection of the HUD for displaying the navigation info.

1.2.2 Scene 2 - Unanticipated Discovery of U-MUSIC Components

After picking up Marc from Duisburg, Thomas tries to plan his route to Keukenhof. But his navigation software, of both the car navigation system and the navigation of the mobile device, do not have license for the Netherlands. Therefore, the route planning to Keukenhof fails. After driving a few kilometers, Thomas arrives at a petrol station near Venlo and stops there for a break and to see if he can get some maps or anything to guide him towards Keukenhof. The petrol station provides an open and free access to a WLAN network (intranet without free access to the WWW) and Thomas's device discovers that another device in the network from another tourist, named Stephan, is also U-MUSIC-enabled and runs an adaptive application. Stephan is using it only for listening to some music; but his application provides a component to download Google maps from the internet. He has a UMTS flat-rate connection and therefore no extra cost

[2] The thesis extends the works from the MADAM and the MUSIC projects in order to support unanticipated adaptation. However, such extensions are not needed for all parts (components) of the middleware. U-MUSIC refers to the extended middleware, while MUSIC refers to the part that remains unchanged.

is incurred for this download. The UnanticipatedTravelAssistant application is configured to use the map downloader component from Stephan's device and after downloading the map it is stored on the local device (Thomas's device).

Thomas wants to examine the downloaded map more closely and while doing it on the mobile device Thomas gets access to a coffee machine at the petrol station. The coffee machine is a smart one ☺ and it provides a large touch screen. After Thomas gets access to the machine, the map viewing task is delegated to this more convenient screen. A printer is also connected in the network and it prints out important route information for a small charge payable using the coffee machine. Afterwards, he gets in the car and leaves the petrol station. The application starts to reconfigure itself to a variant that uses the screen of the car computer and does not need the map downloading component any longer.

1.2.3 Scene 3 - Use of the Online Ticket Facility

Thomas arrives at Keukenhof without any more trouble. He finds the parking place easily, and after parking the car he walks towards the main entrance of the tulip garden. But before entering the garden, he must buy a ticket. However, he notices that the queue in front of the ticket counter is quite big and it may take a long time to buy the ticket. Fortunately, the garden along with its surrounding is provided with a WLAN network and his device discovers an online ticket selling service, which supports buying tickets using the mobile device. It also offers the possibility of buying parking tickets. There is a small entrance having a ticket-checking interface; Thomas's device can automatically detect and exchange data with the ticket-checking interface for ticket validation and the entrance door opens automatically for Thomas to enter the garden without having to wait in the long queue.

1.2.4 Scene 4 - In the Tulip Garden

In the tulip garden Thomas starts taking pictures with his mobile device, which is equipped with a digital camera. He would also like to take a video of the garden, but unfortunately, his device does not support that functionality. However, his device can receive video stream and therefore, he enables the stream-receiving functionality. His device discovers another mobile device that uses the U-MUSIC middleware and provides a video stream service to others. Thomas's application reconfigures itself to add the video streaming component from the other device and stores the accepted stream to the local device. The streaming bit rate depends on the context and resource situations; for example, the connectivity, the amount of free memory in the target (local) device, the number of clients of the stream, the processing capability of the stream provider's device etc.

Alongside the energy-hungry wireless communication, receiving the stream and storing the video data also consumes a lot of energy and the battery power of the device is greatly reduced. Thomas is far away from his car and therefore, he can not recharge the battery. Consequently, the device reconfigures the application so that it stops taking pictures, which requires a lot of battery power for flashes (some areas inside the garden are rather dark, especially because it is a cloudy day). Therefore, it searches for pictures already taken by other people from the locations in the garden that are not already visited by Thomas. It turns out that there is a huge amount of pictures available and because of the limited storage capacity of his device, only a limited number of pictures can be selected. Thomas does not have enough time to sort all those pictures manually based on some characteristics like the object of the picture, quality, picture size etc. and then to select the ones that he wants to transfer; but his device is not capable of sorting images anyway. However, because of the large amount of tourists with a handful of them using U-MUSIC applications, a sorting component is discovered in another device, which can be used transparently to Thomas. Sorting and selection is done based on the picture meta-information regarding the picture quality and the tags. Thomas also decides to upload a few of the pictures to Twitpic in order to share them with his friends. His application has a component for uploading the images automatically to Twitpic or other social networking facilities like facebook, flickr, panoramio etc.

1.2.5 Scene 5 - Return Trip to Germany

Thomas leaves the tulip garden and comes back to his car. He wants to have some traffic information on his way back to Kassel. Thomas indicates to his device that he wants to use a radio and one is provided by the car computer. He does not know the radio stations in the Netherlands to receive the traffic information broadcasted in a language that he understands. His mobile device knows from his user's profile that he understands German and English and therefore, it automatically selects an interesting radio station providing traffic information in the preferred languages.

He still has the coverage of the WLAN provided by the tulip garden authority and wants to find a nearby restaurant to have a meal on the way. Based on Thomas's user profile containing the food habits and also considering the time of the day and that he is travelling, the device selects a suitable restaurant from the list of available advertisements on the net. The restaurant is quite busy and an early order will be helpful to save time. Thomas orders from the menu provided on the net. He also makes an online payment using his credit card.

After having the meal, Thomas leaves the restaurant and drives back home.

1.3 Scenario Analysis

The scenario presented in section 1.2 can be closely examined to find the adaptive behavior of the UnanticipatedTravelAssistant application that supports Thomas during his travel to Keukenhof. A rigorous analysis would find all the context and resource dependencies, discovery of new services, devices and components along with the adapted configuration of the application. However, in the analysis present in Table 1, I focus on all the adaptation events and look for the possible reasons that trigger such adaptations. In the process, I also highlight the level of anticipation, identifying if the complete or part of the adaptation could be foreseen at design time or it is completely unanticipated until the adaptation reasoning time. I also point out where new requirements are added to the application, which also triggers adaptation through finding new components and services corresponding to the changed requirements.

Table 1: Analysis of the scenario

Scene	Adaptation	Level of Unanticipation
1 Traveling to	a. Upon the detection of the car computer, its more convenient navigation system with the	a. Semi-anticipated adaptation. The car navigation system can be considered as a

the Netherlands	broader screen is chosen to present the navigation information. The stored data on the mobile device is available to this system. This also saves the battery power of the mobile device.	service, the need for which can be foreseen by the developer of the UnanticipatedTravelAssistant application; but he is not aware of the realization. Also, the developer of the car navigation system does not need to know about the developer of the UnanticipatedTravelAssistant application.
	b. While driving hands are busy, the voice command interface is automatically activated for providing input to the mobile device.	b. Anticipated adaptation, because the need for and the realization of the voice command facility can be foreseen at design time.
	c. A busy street with a high traffic rate triggers the use of the HUD for presenting the navigation information.	c. Semi-anticipated adaptation. A HUD is considered as a service.
2 Unanticipated discovery of U-MUSIC components	a. Downloading maps using the component from Stephan's device and then storing the downloaded map to Thomas's own device.	a. Unanticipated adaptation, because the availability of Stephan's device and the map downloader component can not be predicted earlier. Here we consider that the applications on these two devices are developed by independent developers who do not know about each other's development.
	b. Viewing the map on the touch screen of the coffee machine. The access of the machine is detected and only then it can be used. Using the printer to print out route information.	b. Semi-anticipated adaptation. The touch screen and the printer are considered as services.
3 Use of the online ticket facility	a. Buying the online ticket, when they are available on the net.	a. Semi-anticipated adaptation. Online ticketing facility is considered as a service.
	b. Detection of the ticket screening service and exchanging data automatically to the device for	b. Semi-anticipated adaptation. Online tickets can be checked either manually (no

	ticket validation.	adaptation) by some screening interfaces, or automatically (semi-anticipated adaptation) as described in the scenario.
4 In the tulip garden	a. Finding a U-MUSIC-enabled device which provides a video stream. Using that component, store the video in his device. Thomas may have exclusive control to the video recorder or he may only receive the stream provided by it.	a. Unanticipated adaptation. The presence of the video taking component is not foreseen, its need is not specifically expressed; but it can still be used because it is U-MUSIC-compliant. Also, note the update of the application's functionality/ requirement by the user at runtime.
	b. Upon reduction in the battery power, stop taking images and searching for pictures already taken by others.	b. Anticipated adaptation. The picture taking capability clearly depends on the battery power.
	c. Using the sorting component from a discovered U-MUSIC-enabled device and selecting a number of pictures for sharing.	c. Unanticipated presence of U-MUSIC-enabled device providing the sorting and selection support.
	d. Uploading the selected pictures to Twitpic. There are other options like facebook, panoramio, flickr etc.; but uploading to Twitpic is preferred, based on the ease of uploading (site speed), presence of online friends etc.	d. The selection of the proper site can be anticipated. However, this adaptation is triggered by adding the new functionality of uploading the photo.
5 Return trip to Germany	a. Selecting a suitable radio channel based on the user's profile.	a. The choice is semi-anticipated, because the user's profile is already known, although the radio application is developed independently of the UnanticipatedTravelApplicati on application. This requirement is added at runtime.
	b. Selection of a nearby restaurant	b. The selection and availability

	based on the user's profile and the address of those restaurants. Thomas's location is known from the address of the tulip garden.	of proper restaurants can be semi-anticipated, because the user's profile is known.

From the scenario analysis presented in Table 1 it is evident that some of the adaptation possibilities can be foreseen at design time, while some of them can only be partially foreseen and depends on the availability of services at runtime. However, some of the adaptation possibilities can not be foreseen until the application is running. Having a closer look at the analysis, we can see that the use of third party services is mostly considered as semi-anticipated, because the need for service must be anticipated at design time, while the developer does not need to know about the actual realization.

However, realizations using U-MUSIC components mainly fall in the category of either anticipated or unanticipated adaptation. When the usage and presence of a particular component is obvious at design time, it is considered as anticipated. On the other hand, when the application realization depends on the components provided by other developers, especially when no information about their realization details can be foreseen, this leads to the unanticipated adaptation. U-MUSIC applications may be realized by components developed by a number of independent developers. In order to distinguish between anticipated and unanticipated adaptation, we need to consider if a developer knows which application[3] his component will realize.

Also, the scenario presents cases where some functionalities or requirements can be added or removed at runtime. Such dynamic changes in the requirements may be both anticipated and unanticipated. For example, when some functionalities of the application are designed as optional, they can be selected either manually or automatically at runtime. However, they exist since the design time, and therefore they are anticipated. On the other hand, an example of unanticipated changes in functionalities or requirements can be when the user can add new functionalities at runtime. The basic idea of such unanticipated adaptations is to provide the user with the best possible support, based on all the scenarios – consisting of the availability of services or devices along with context and resource situation –, whether such scenarios can be thought of beforehand or not.

1.4 Challenges

The support of context awareness and self-adaptation of mobile applications, in general, offers a number of research and development challenges. Adaptive mobile applications need to be supported with sensing the context, discovering devices and services and collecting necessary information about them, reasoning on the information to take the adaptation decision and then reconfiguring the application based on that. Things get more complicated because of the usual resource shortage of a mobile device. The introduction of the unanticipated adaptation introduces new challenges, because it

[3] Here application refers to an application type. In technical terms, the realization of an application type is considered as an application. In general, a component realizes a component type. An application type is a specialization of a component type.

requires adaptation to the scenarios that may be completely unforeseen during the application development. In the adaptation scenario presented in section 1.2 we have to confront various adaptation problems that may be of the type anticipated, semi-anticipated or unanticipated. Keeping the analysis of the scenario in section 1.3 in mind we can derive a number of requirements as well as challenges faced while addressing the adaptation of mobile applications. For each of these challenges, the type of adaptation is also investigated. Moreover, it is clarified in which extent each of the challenges will be addressed in this thesis.

1.4.1 Application Variability

Adaptation may be achieved in numerous ways; for example, the configuration parameters of some components may be changed, some components or services may be added or discarded, some components may be relocated in a different node etc. Such actions eventually create a different variant of the application, maintaining the core functionalities and changing its quality of service or adding or discarding a few functionalities, with the availability or unavailability of new components and services.

The idea of self-adaptation is to integrate some adaptation capabilities within the application architecture. This can be achieved by introducing a variability model, based on which different application variants can be created at runtime. For the anticipated adaptation, especially if all the components, their QoS properties etc. can be foreseen at design time, the application variability model can be statically defined. However, this can be enhanced dynamically adding new components at runtime. The developers of such components may or may not have prior knowledge of the application their component will be used to realize. Therefore, such dynamic update of the variability model applies both for anticipated and unanticipated adaptation. This also applies for semi-anticipated adaptation, because discovered services can also be used as alternatives to components. This thesis addresses this challenge profoundly.

1.4.2 Inter-operability and Heterogeneity

A close look at the scenario of section 1.2 will easily reveal that the developers of adaptive applications will find it quite difficult to support inter-operability between applications and services developed independently by different parties. For example, Thomas and Stephan had two different U-MUSIC applications, which are most likely developed by two different sets of developers. Ensuring that some components from one application can be used by some other applications is a challenging task, especially when one component developer does not have any prior knowledge of the application type (or component type, in general) his component will be used to realize. In the case of unanticipated adaptation the developers should get the freedom of focusing on the development of their own applications and components, which are compliant to the U-MUSIC middleware, without having to know about the development of others.

The same is true for third party services. Services in ubiquitous computing environments may have been developed independently by different actors and organizations. This implies that services and their QoS properties may have different names and representations. Likewise, heterogeneity may be found within the context management system, in particular if third party context sensors and reasoners are integrated. Thus, QoS properties and context information that describe the properties of the execution context require semantic annotations in order to enable interoperability and integration.

This thesis addresses the problem in providing an ontology-based modeling approach in order to provide a common vocabulary. The ontology is extensible so that individual component and service providers may define their own ontology based on the common vocabulary.

1.4.3 Dynamic Discovery of Devices and Services

In a ubiquitous computing environment, new services and devices may become available or unavailable in the adaptation domain[4] without any prior notice. For example, in the scenario, Stephan's device was available in the petrol station; the ticket selling service, the video recorder, the image searching and sorting components etc. have been available at the Keukenhof garden. In general, there can be hundreds of such devices or services. The device running the adaptive application should be able to discover them and use the provided components and services as needed for its own application. Such discovery should be done at runtime, while the application is already operating. Moreover, the discovery process should be transparent to the users.

This challenge applies to all three types of adaptation – anticipated, semi-anticipated and unanticipated – and it is addressed in this thesis.

1.4.4 Dynamic Updates of Requirements

When an adaptive application is running, the user may want to update *what* he wants from the application. Therefore, the requirements (e.g., the expected functionalities) of the application may be changed seamlessly without needing to stop the application. Such requirements may be thought of at design time. For example, there can be some core requirements needing to fulfill all the time, while some requirements - both functional and non-functional - may be optional. The need for such optional requirements may be specified by the user himself or they may be activated based on the context situation. In support of the unanticipated adaptation, we would like to provide flexibilities so that it may be possible to add new requirements by the user even at runtime of the application.

In this thesis I address this challenge in some extent, mentioning some possible solutions. However, no implementation is provided yet.

1.4.5 Context Sensing and Reasoning

When a user is moving around in a ubiquitous computing environment, he will face changes in the state of the computing environment. Moreover, the device capabilities also change with time and its usage. For example, in the scenario, the storage capability changes, the battery power reduces etc. Also, the location of the user restrains the access of the GPS service; a cloudy weather requires the use of flashes for taking pictures etc. Adaptation is required corresponding to such dynamic changes of the context.

[4] An adaptation domain is a collection of U-MUSIC middleware instances controlled by one adaptation manager. It includes one MASTER node (normally a handheld device) which represents a binding to a user and acts as the nucleus around which the adaptation domain forms dynamically as SLAVE nodes come and go.

In order to adapt the application to such context changes, the appropriate context information must be retrieved, requiring the usage of context sensors. Moreover, such context information is often raw and needs some post-processing to use them in the adaptation reasoning mechanism. Context reasoning refers to this post-processing process. Most often the application developers are the ones to identify which context information is needed for their applications and components and they have to provide appropriate context sensors and reasoners. In a ubiquitous computing environment, sensors developed by different developers may have different representations in their context data. This also poses a challenge to successfully using the sensor data.

This challenge is addressed in the MUSIC project and I use that support without descibing them extensively in this thesis. For a detailed description of how it is supported in MUSIC, please refer to the MUSIC WP2 deliverables [12] [13] and the doctoral thesis of Paspallis [89].

1.4.6 Adaptation Reasoning - Performance and Scalability

When new services and devices are discovered or a significant context change occurs to deteriorate the performance of the application at runtime, the need for an adaptation is triggered. The task of adaptation reasoning involves finding a variant of the application that best fits the current context, when such an adaptation need is triggered. Upon selecting the best-fit variant, the application is reconfigured. Such a process may impose a big challenge in terms of its performance, especially for mobile devices with limited computation resources.

Ideally, an adaptation through reconfiguration should happen in a blink, such that in terms of performance the user will not notice the adaptation activities. While a reaction to a changed sensor value might be accomplished in this manner, discovering a service, performing service-level negotiations and binding a service via a proxy will, in most cases, take at least a few seconds, if not more. Whether this is acceptable to the user of the application depends very much on the application scenario as well as on the degree of interactivity of the application's user interface. The same restrictions also apply for discovering U-MUSIC-enabled devices and using their components.

During the adaptation decision, resolving all possible variation points and considering all possible realizations can effectively create a huge number of different application variants, all of which must be evaluated for their utility in order to find the one with the highest utility. Obviously, for a high degree of variability this reasoning approach will suffer from combinatorial state space explosion. This is a general concern in self-adaptive systems. With the service-based and unanticipated adaptation we need to exert even more concerns for scalability. Service-level negotiations with too many service candidates would certainly lead to scalability problems. Furthermore, it is a waste of time and resources to take services that will not be available for long into account during adaptation planning. For the unanticipated adaptation, the number of possible variants is completely unknown until the adaptation reasoning starts. Therefore, the adaptation reasoning algorithms need to be as stable as possible against the increase in the number of variants, so that a meaningful adaptation can be offered to the user within a reasonable time, irrespective of the number of possible variants.

This thesis proposes a new adaptation reasoning algorithm to confront this challenge.

1.4.7 Robustness

The fact that devices and services may appear and disappear at anytime implies that adaptations that employ more than one device or use services are vulnerable to failures during the adaptation process. Some components or services may be unavailable while they have been performing some tasks; for example, the video stream provider component in the scenario may be unavailable or Stephan may leave the petrol station before the map downloading is finished. In that case the application should reconfigure to a working state, deciding if the already available data may be usable or not. The situation is even more complicated if some devices or services leave the adaptation domain just after they are chosen by the adaptation reasoning process and therefore they become unavailable during reconfiguration. The application may not go back to the earlier configuration – it was unsuitable anyway, triggering the adaptation – and may be the adaptation process needs to be restarted. When a service is used, it has to be observed if the negotiated quality of service is maintained. All such considerations refer to a complex system, which has to act intelligently and usually quite rapidly, while retaining its usability.

In MUSIC, some researches are going on to work on this challenge. However, no definite solution is reached yet. This thesis does not address this challenge either.

1.4.8 Testing and Validation

Testing the functional correctness of context-aware and adaptive applications in general is inherently difficult. Not only do we need to test the application logic itself, but also the reactions to context changes. Depending on the number of involved context sensors, related context events, and number of potential service bindings, testing can be a very complex and demanding task. Testing should start as early as possible in the development process.

In the case of the unanticipated adaptation, the problem is even more complicated, because the application may come across situations – context, resources, availability of devices and services – that can not be foreseen beforehand. Therefore, the support of the unanticipated adaptation requires more general solutions that would work correctly irrespective of the variations from ideally expected situations.

In the MUSIC development methodology [64], we provide a number of suggestions to the developers in order to test and validate adaptive applications in general. This thesis extends that work to some extent. MUSIC is also working on providing a toolset that can be used for that purpose. In MUSIC, we are developing a number of trial applications, which support anticipated and semi-anticipated adaptation. In this thesis, I have tested the correctness of the current implementation of the U-MUSIC middleware in supporting unanticipated adaptation. I have also evaluated the performance of the adaptation reasoning algorithm. However, no real-life application is developed or tested for this thesis.

1.4.9 Usability and Security

The aspects of ergonomics and usability play an important role for the acceptance of any user-oriented IT system, and especially for self-adaptive systems. Too many user-visible adaptations will disturb the user. So, the question arises how much adaptation activities can be inflicted upon the user. In addition, criteria such as controllability by the user, self-explaining adaptation activities, and comprehensibility are important concerns that may partially be in conflict with goals such as transparency and reduced user interactions. Furthermore, the question of trust is important so that we can increase the user's level of trust in the adaptive system.

All these questions are particularly difficult for the service-based and unanticipated adaptations. In a multi-user scenario as presented in section 1.2, the interactions of different users can not be foreseen and therefore, such interactions may be unexpected in many situations. For example, in scene 2, Stephan might not be willing to provide the map downloading facility or in scene 4, the streaming or image sorting facilities may be offered only to trusted clients. From a client's perspective, Thomas also has to get a clear idea whether any security threat is involved when he is going to use a service or some components from other providers.

There are some ongoing research activities in MUSIC regarding this challenge. However, there is no explicit solution available as yet and this thesis does not address this problem either.

1.5 Focus and Contribution

I have been working in the MADAM and the MUSIC projects for the past few years. Obviously, this thesis adopts the developments in those projects. However, those projects do not explicitly address the unanticipated adaptation. Therefore, the contribution of this thesis can be viewed as the support to the unanticipated adaptation in addition to my work (in teams with others) in those projects. However, this thesis focuses on the unanticipated adaptation problem in a way to introduce the new developments as an extension to our works in those projects, whereas contributions to those projects are either referenced or briefly presented for completeness.

In reference to the definitions provided in section 1.1, the MADAM project basically provides a solution for anticipated adaptations. MUSIC succeeds MADAM adding the support for semi-anticipated adaptations facilitating the use of third party services as alternatives to components in realizing adaptive applications. For both projects, the main results include the conceptual development on context awareness and self-adaptation, a middleware platform for supporting adaptive applications, a model-driven development methodology for the application developers as well as for the researchers and a set of tools that supports the development process. This thesis adopts those results and makes necessary updates in the aim of adding support for unanticipated adaptation.

In the following, I provide a list of main extensions that I have made in comparison to the MUSIC results:

- **Conceptual meta-model:** The conceptual meta-model is extended to support the unanticipated adaptation-related concepts that help to build the application variability model at runtime. It is also simplified by discarding the role concept. Moreover, the port type concept is used only to indicate interaction points.

- **Adaptation reasoning approach:** A new reasoning approach is developed to support adaptation reasoning even when the number of application variants is quite huge. I also show that the reasoning time is not much influenced by the increase of the number of application variants. The complexity (linear) of the approach is compared with that of the existing MUSIC solutions.

- **Middleware architecture:** Four components, namely Bundle Manager, Adaptation Middleware, Repository and Information Model, are updated in the middleware architecture. The Information Model copes with the changed data structure corresponding to the updated concepts, the Bundle Manager supports runtime matching of types and realization plans, the Adaptation Middleware integrates the new adaptation reasoning approach and the Repository for registering discovered bundles and their artifacts is adjusted to register plans and types correctly.

- **Middleware implementation:** Corresponding to the extensions in the middleware architecture, an initial implementation is provided.

- **Development methodology:** Some of the steps of the methodology for developing unanticipated adaptive applications are updated. The domain modeling is enhanced by adding a Functionality Ontology, the variability modeling allows independent development of types and plans, and the transformation methodology is updated through the code completion technique corresponding to the changed approach of specifying utility functions.

- **Tools:** The transformation tool is updated in relation with the extensions in the modeling methodology.

Details of those extensions will be presented throughout the document in connection with each of the topics.

1.6 Document Structure

The document is divided in three main parts: the first part provides all the background information needed to understand the remaining of the document, the second part describes how the problem of the unanticipated adaptation is solved and the last part evaluates the solution and discusses its pros and cons identifying the scope of improvements. Part I has three chapters, part II has four and part III is divided in two chapters. In addition to these three parts, other relevant information is presented in appendices. The contents of the rest of the main chapters are summarized as follows:

Chapter 2: This chapter provides some background information discussing different adaptation approaches and policies.

Chapter 3: This chapter provides the state of the art in related fields. I first discuss the work done in the field of context awareness and self-adaptation in general, and then introduce what is done in MADAM and MUSIC, the projects that are used as the baseline for this thesis. Afterwards, I discuss a few works in the direction of unanticipated adaptation.

Chapter 4: It presents the basic concepts used to address the problems of the unanticipated adaptation. A conceptual meta-model suggests the relations among different concepts. Then it is described how different application variants are created according to the conceptual meta-model. It also discusses a few adaptation policies and the rationale behind using the utility-based policy for this work.

Chapter 5: This chapter describes the adaptation mechanism, clarifying the deployment of application bundles, construction of application variants and reasoning of adaptation including the newly developed adaptation reasoning algorithm.

Chapter 6: The middleware is described in brief in this chapter. The MUSIC middleware is adopted as the baseline and four of the middleware components are updated to support the unanticipated adaptation. Those updated components are described in detail, while the rest of the middleware is either referenced or presented briefly.

Chapter 7: This chapter provides the guideline to the application developers along with a description of tools that they need to use during the development process. The MUSIC methodology, which consists of a number of steps guiding the application development, is adopted as the base line and therefore, this description mainly highlights the updated steps, while all other steps are described only briefly. Tools are also presented in brief.

Chapter 8: This chapter describes two test applications in order to validate two features: 1) the support for the unanticipated adaptation and 2) the performance of the adaptation reasoning approach for large scale applications. These are no real-life applications, because the functionalities of components are not implemented. However, they are designed to verify the features as mentioned.

Chapter 9: Finally, chapter 9 discusses the work done in this thesis providing an insight to what is done and what else can be viewed as possible improvements in the future.

2 Adaptation Concepts

Adaptation can be achieved by applying a number of different approaches; e.g., in some cases, adaptation may trigger changes in the application composition, while in some other situations, configuring some components with a different set of QoS properties does the trick. Moreover, there are different policies to reason on the context changes and take adaptation decisions. This chapter introduces the concepts related to some of these approaches.

2.1 Context Awareness and Self-adaptation

The concepts of context awareness and self-adaptation are often sources of confusion, because self-adaptive applications often adapt their behavior based on the context stimuli and therefore, it is often difficult to make a clear distinction between these two concepts. Paspallis [89] has clarified the concepts from the perspective of the functional and the extra-functional behaviors of an application. A purely context-aware application is identified as using the context information simply to complement its functional behavior. On the other hand, a context-adaptive application adjusts its extra-functional behavior based on the context information. As an example of purely context awareness, a mapping application running on a mobile device is mentioned. Such an application uses the location information to automatically center the map at the current location of the user. However, that map may have different views, e.g., normal street map, satellite view etc. and such views may be chosen based on the device resources, user's needs, user's motion etc. In this case, the application is termed as context-adaptive.

From the perspective of the user's interaction with the application, Paspallis [89] divides adaptation in two categories: Some applications are self-adaptive and some others are explicitly adapted by external actors such as users. All self-adaptive applications can be viewed as context-adaptive, because they use context information for adapting their extra-functional behavior.

In this thesis, I adopt the definition from Paspallis [89] and I mainly focus on providing self-adaptation to mobile applications. However, I also provide support for adding new functionalities as well as adjusting the set of functionalities by the user at runtime.

2.2 Adaptation Approaches

McKinley et al. [3] mention two general approaches in realizing dynamic adaptation to software: parameterized adaptation and compositional adaptation. In addition to this, adaptation can be realized by weaving aspects [90] [91] at runtime to the base components depending on the context situation. In the following I provide some insight in these approaches.

2.2.1 Compositional Adaptation

Compositional adaptation refers to the exchange of algorithmic or structural parts of a system in the aim of fitting it to the current environment. Such adaptation approaches are particularly useful in the case of component-based application, where the application is considered as a composition of components, and alternative component implementations are used to realize particular functionalities of the application. Choosing among the alternative components based on the current context, a particular composition of components is used to realize the application. The approach remains valid with the integration of services as alternatives to components in realizing application functionalities.

Compositional adaptation is well-suited for unanticipated adaptation, because it enables an application to adopt new components and services for realizing application functionalities unforeseen at design time. They can easily cope with the resource shortage, changes in the context, and availability and unavailability of components and services at runtime.

2.2.2 Parameterized Adaptation

Parameterized adaptation involves the modification of variables that determine the program behavior [92]. In the case of component-based application, some behavior of a component can be dependent on certain parameters and therefore, instead of replacing the component itself, that behavior is adjusted adopting a different parameter value. When a number of parameters govern the component's behavior, sets of values can be defined.

Parameterized adaptation approach is often applied for fine-tuning an application's behavior without making any structural changes. Parameter settings are defined at design time based on some ranges of values. For practical applications there are some constraints on choosing such value ranges. For example, a continuous value range would effectively create an infinite number of parameter settings, eventually making it impossible to evaluate the appropriateness of all the settings. Choosing concrete values for parameter settings solves that issue. Another shortcoming is that it cannot adopt algorithms or components left unimplemented during the original design and construction of an application [3]. Therefore, parameterized adaptation is not very well-suited for unanticipated adaptation.

2.2.3 Adaptation by Aspect-weaving

The adaptation approach by aspect-weaving integrates the principles of aspect-oriented programming (AOP) [93] in the development of software components. AOP supports the construction of reconfigurable systems by enforcing separation of concerns. Complex programs include various crosscutting concerns, e.g. QoS, energy consumption, fault tolerance, logging, security etc. [93] An aspect is defined as a set of pieces of code (advices) to execute in particular points (pointcuts) of an application. Pointcuts are usually composed of a set of elements of the base code (joinpoints such as class, method or control instruction). Aspect weaving is the mechanism that inserts the aspect advices into pointcuts at compile-time, load-time or runtime.[90]. Thus the developers can isolate the implementation of identified aspects from the base component. Then, aspects are combined to the base component to automatically produce new component implementations. In addition, the isolated aspects can, in some cases, be reused and combined with different component implementations. The adaptation is achieved by selecting the combination of component implementation and aspects providing the best QoS to the user depending on the current context.

Aspects can enhance an application by introducing new functionalities, as well as by improving an existing functionality. This approach of adaptation can be used for both anticipated and unanticipated adaptation.

2.2.4 Adopted Adaptation Approach

In the research projects MADAM and MUSIC, we have supported both parameterized and compositional adaptation, where compositional adaptation is the heart of our development, where parameterization also has found some limited use. However, in MUSIC, the partners from the University of Oslo have been working on the adaptation approach by aspect-weaving. [10][90]. This is still work in progress and the concepts developed in the researches have not been implemented yet in the middleware. This thesis addresses only the parameterized and the compositional adaptation approaches.

2.3 Adaptation Reasoning Policies

Adaptation reasoning policy defines the criteria that are used to select the best-fit application configuration among different configuration possibilities. Depending on the system and the targeted domain, different adaptation policies can be adopted. A few of such policies include action-based or rule-based adaptation, goal-based adaptation, utility-based adaptation etc. These policies are discussed in details in the MADAM deliverable D2.2 [10] and in the following I introduce them in brief.

2.3.1 Action or Rule-based Adaptation Reasoning

Action-based policies have been quite popular and are used in different domains related to networks and distributed systems such as computer networks, active databases and expert systems. An action policy consists of situation-action rules which specify exactly what to do in certain situations. Some authors such as [46] have considered policies for controlling networks and distributed computing systems that are based on such situation-action rules.

In the domain of software architectures, the concept of event-action rules has been used to express and manage the dynamics of systems' architectures. These rules may be expressed at the ADL (Architectural Description Language) level by associating invariants in the form of event-action rules in order to model component behaviors that are projected to the runtime level. At the Workshop on Architecture Description Languages [47], rules were presented to express dynamic reconfigurations over component-based architectures.

Some more focused and specialized works on policies for network and distributed systems such-as [48] and [49] propose an action-based policies approach based on the event calculus in order to deal with the adaptability in mobile and pervasive computing. These works are more elaborative in the sense, that they provide languages and policy engines to handle adaptability and cooperation between different competing applications and users. However, these works are very complex and still in their infancy without real demonstrations.

While the action-based approach appears to be powerful, in pervasive computing the management of such action-based policies becomes complex from the user point of view. Indeed, action-based approaches require policy makers to be intimately familiar with low-level details of system function – a requirement that is incompatible with the long-term goal of elevating human administrators to a higher level of behavioral and

QoS specification. In other terms, this approach does not consider the mapping between different levels of interests. Also, it becomes very difficult, if not impossible, to apply this technique for systems supporting the unanticipated adaptation, because rules and actions can not be foreseen in such cases.

2.3.2 Goal-based Adaptation Reasoning

Goal-based adaptations are a higher-level form of behavioral specification that establishes performance objectives, leaving the system or the middleware to determine the actions required to achieve those objectives. This is typically the case of some works that determine algorithms that allocate and control computational resources to guarantee promised levels of QoS. Since goals provide only a binary classification into “desirable” and “undesirable” performance, works in goal-based adaptation concentrate much on maximizing the probability of achieving goals or minimizing the degree to which goals are not met [50][51].

Also, there are many works in AI and especially in the context of multi-agent systems and early-planning algorithms [52]. In some multi-agent systems, autonomous agents may be goal-oriented, having social abilities to communicate with other agents. The cooperation between individual agents converges and tends to achieve the global application goal.

Finally, a general lack with the goal-based approach is that solutions are classified in a binary way – “desirable” and “not desirable” – without offering mechanism and flexibility to measure how one solution is appropriate to one situation in order to be able to negotiate contracts between competing mobile adaptive applications.

2.3.3 Utility-based Adaptation Reasoning

Utility-based adaptation permits, on the fly, the determination of a ‘best’ feasible state while goal policies place the system in any state that happens to be both feasible and acceptable with no drive towards further improvement [9].

Many works use utility functions to qualify and quantify the desirability of different adaptation alternatives. Most of these works are QoS-based, applied in different domains for resource allocation [53] and typically in mobile and pervasive systems such as Odyssey [53] and QuA [55] . In most of these works, utility functions are usually specified directly in terms of resources and QoS dimensions. As very close to the principle of the utility function, Odyssey introduced the Principle of Fidelity to measure the degree to which a data item available to an application matches a reference copy. The ideal data copy is one that does not consume resources. In QuA, utility function is used to determine the desirability of different implementation alternatives of a service as a function of its QoS.

2.3.4 Adopted Reasoning Policy

In this work I have used a utility-based approach, where the term ‘utility’ is introduced as a measure of how well a software system fits a given context. The utility is given as a function of the QoS properties of a particular realization of a component type, indicating the deviation from a perfect case; i.e., a comparison between the properties expected by the system and that provided by its context.

The value of the utility is calculated during the adaptation using a utility function, and a comparison among the utilities of different variants of the application help to select a particular realization. In order to facilitate such a selection, in this work we propose that the utility value needs to be normalized within the limit of 0.0 and 1.0. It facilitates adopting any format of the utility function as long as the value is ensured to remain within this range.

Unlike MUSIC my approach supports the concept of part utilities so that the overall utility of a composition can be obtained from combining the utilities of the constituents. Such a combination may be considered in the same way as combining properties in a utility function. A straightforward format of the utility function may be a weighted sum of the differences between the expected properties and properties provided by the context. For a utility of a composition, weights can be assigned to part utilities. However, the utility-based approach is not limited to such simple form of utility functions only.

From my experiences in the MADAM and the MUSIC project, where we supported a number of developers in developing proof of concept applications, it was found that assigning appropriate utility functions is quite difficult, especially when the developer has to think about the complete application and its fitness to different context situations. The problem becomes more difficult for the unanticipated adaptation that we support in this work, because of the flexibility allowed to the developers: In extreme cases they can just express their needs of *functionalities* to be performed by the application, and the rest can be decided automatically at adaptation time, based on the availability of components and services providing such functionalities. On the other hand, the component developers or service providers may not have any concrete idea which application their components or services will be used for. Therefore, they can not provide any application-specific utility function. Such challenges have motivated me to provide a new adaptation reasoning technique, which provides the facility that an individual developer may focus on the utilities of the components developed only by him, without bothering what the others are developing.

3 Related Work

The topic of context awareness and self-adaptation, in general, has been of great interest to the research community for several years. Like the MADAM and the MUSIC projects, this thesis also supports self-adaptation to mobile applications. Such self-adaptive applications are most often influenced by the context situation and therefore, they can be termed as context-adaptive as explained in section 2.1. This thesis certainly includes the work done in the MADAM and the MUSIC projects, while those works are extended in the aim of providing a solution to the unanticipated adaptation problem. Therefore, MADAM and MUSIC works are mostly referenced or introduced only in brief and I focus on the unanticipated adaptation.

With that aim in mind, I first introduce the main results obtained in the MADAM and the MUSIC projects. Afterwards, I mention some works done in the area of context awareness and self-adaptation, comparing them with the solution provided in MADAM and MUSIC as well as in this thesis. We have discussed such works in broader extent in MADAM and MUSIC deliverables; for example, D2.2 [10] of MADAM and D1.3 [11], D2.2 [12], D2.3 [13] of MUSIC. This thesis only discusses the most relevant ones in a more compact manner.

At the end, I describe several works that introduce the unanticipated adaptation problem. In the process, I discuss the different definitions of unanticipated adaptation as adopted in different related works, the challenges and solutions of unanticipated adaptation, presenting a comparative study of those works in relation to the solution provided in this thesis.

3.1 MADAM and MUSIC

The *Mobility and Adaptation-enabling Middleware (MADAM)* project [1] has addressed adaptation from both the theoretical and the practical perspective and solves a number of challenges [94]:

- Adaptation happens seamlessly and without user intervention in reaction to context changes.
- Applications may exploit any kind of context dependencies as long as there is appropriate hardware and software available in the computing environment to provide these context data.
- Context awareness and adaptivity of applications are treated as a separate concern in application design.
- A general component model and middleware infrastructure support many adaptation styles, e.g. local and distributed adaptation, parameter and compositional adaptation.

- A model-driven development approach comprising adaptation models and corresponding transformations facilitates the development of self-adaptive applications and the reuse of adaptation artifacts.
- Real applications from industry partners are used to evaluate the approach.

In solving those challenges, MADAM provides:

- A conceptual solution to the adaptation problem,
- A sophisticated middleware that supports the dynamic adaptation of component-based applications,
- An innovative model-driven development methodology which is based on abstract adaptation models and corresponding model-to-code transformations, and
- Two real-world trial applications to demonstrate the viability of the MADAM solution.

However, a MADAM application is completely component-based, whereas new technological achievements have introduced additional requirements and opportunities. For example, ad-hoc networking facilities and ubiquitous service architectures are made available that represent an enrichment of an application's execution context. Thus, an adaptive application may want to replace a local component by a remote service if it promises a better quality. Using services becomes more prominent, because context-aware self-adaptive applications are most often distributed applications running on a ubiquitous computing environment, where services play a major role. In order to support the integration of services, we need to face new challenges; for example, new context models and context query languages are needed to model these environments and fully exploit such scenarios; the usage of remote services must be controlled by some implicit or explicit service level agreement; adaptation decisions may depend on the quality of a service as well as on its price. Moreover, it has to be explored what kind of decision support techniques are appropriate for controlling such adaptations. With our experiences with MADAM, we have observed that the middleware itself needs to adapted dynamically; for example, the adaptation component of the middleware may be configured to use a particular adaptation reasoning algorithm from a possible set of options, a local or remote reasoner may be used based on the context situation, the context component of the middleware may adapt itself to manage different context types etc. We face some more challenges, when we also allow the middleware to adapt dynamically.

The majority of the MADAM consortium members also take part in the *MUSIC (Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments)* project [2]. In this project we have addressed issues that arise towards supporting dynamic runtime adaptation of both applications and the middleware, with the possibility of integrating third party services in the application architecture. The MUSIC project defines 8 feature groups, where each feature group consists of several features related to a particular topic. The contents of the feature groups are briefly presented as follows:

- **Context sensing and synthesis:** This feature group focuses on generic and application-specific context sensors and reasoners, transparent access to local and remote context in a distributed computing environment, security and privacy of context information etc.

- **Multi-dimensional decision making:** This feature group focuses on making the adaptation decision process automatic. It also covers the development and the validation of property predictors and utility functions that aid in making such adaptation decisions.

- **Compositional adaptation:** The compositional adaptation feature group focuses on adaptation by component replacement, configuration parameter setting and component relocation. It also addresses issues like reliable reconfiguration, state transfer when relocating components, device adaptation, architectural constraints when building hierarchical application variability architecture etc.

- **Reuse and evolution support:** This feature group mainly addresses independent development of components and the dynamic evolution of the application. Multiple applications may run on a device and they can be adapted concurrently.

- **Services in the SOA sense:** This feature group consists of dynamic discovery of devices and services, negotiation and monitoring of service level agreements, hosting services on MUSIC nodes as well as incorporating services into the adaptation reasoning mechanism.

- **Adaptation of the middleware:** This feature group aims at supporting the manual configuration of the middleware, self-organization of the activities of several middleware instances and self-adaptation of the middleware components.

- **Advanced features:** A number of advanced features like the support of multi-user applications, adaptation by dynamic aspect-weaving, learning and reasoning with uncertain context information are addressed in this feature group.

- **Extra functional requirements:** In addition to the adaptation problem, MUSIC targets to support a number of extra-functional aspects like performance, security, scalability, robustness, non-intrusiveness, platform independence etc.

Like MADAM, MUSIC results also consist of theoretical solutions, along with a middleware, a development methodology and three trial applications. However, the set of challenges addressed in the MUSIC project is quite large. On one hand, it vastly improves on the features and challenges addressed in MADAM; on the other hand it adds many new features. However, MUSIC also does not address the problem of the unanticipated adaptation. In MUSIC, a number of different developers may develop their applications and components independently and such components may be re-used. However, the meaning of independent development is limited in this case, because a component can be used to realize only a particular component type and therefore, the component developer must know this type information (defined by himself or another developer) at design time. Moreover, a particular component is limited to realize a certain type only. With a view of supporting unanticipated adaptation in this thesis, I have worked on getting rid of such limitations as much as possible so that the component developer may think independently of which type his component will be used to realize. In this way, he can be truly independent of other developers as he does not need to anticipate what others are developing. I also support the possibility of imprecise matching between types and component realizations so that a particular component may be used to realize a number of different types.

One of the most important research concerns, which are directly related to the device computation capability, is the adaptation reasoning mechanism. Because of the limited computation resources on the target mobile devices, the reasoning approach and algorithms must be efficient enough to provide a reasonable and in-time adaptation. Both MADAM and MUSIC adopt the utility-based adaptation reasoning policy (see section 2.3.3) and develops adaptation reasoning algorithms to find the application configuration that maximizes the utility. MUSIC currently has three adaptation reasoning algorithms with varying degrees of usefulness [57] [106]. All the MUSIC trial applications have a limited number of application variants and those reasoning approaches are sufficient to satisfy their needs. However, the situation gets complicated when the number of application variants increases; for example, with the availability of new services and components to realize particular component type, the number of total application variants increases exponentially, as we will see in section 5.3. The same effect is also observed in the case of multiple applications, where the total number of possible combinations becomes a multiplication of the number of variants of individual applications. Therefore, all the solutions that we have so far in MUSIC suffered from the scalability in some extent.

It is expected that in the case of the unanticipated adaptation, especially when a good number of services and devices are considered in an adaptation domain, the number of possible application variants can not be guaranteed to be within some particular limit. Therefore, the existing reasoning approaches may become useless. The unanticipated adaptation also imposes another challenge that it is not reasonable to define a particular utility function, as it is done in MUSIC, for the complete application. The application developer may not foresee the components that will be used to realize his application and therefore, the dependencies to context and resources are revealed only at runtime, based on the available components and services. In this thesis, I have presented a new adaptation reasoning approach that takes into account those challenges to provide an effective solution.

3.2 Context-aware Self-adaptation

All self-adaptive applications can be viewed as context-adaptive, because they use context information for adapting their extra-functional behavior [89]. This thesis also addresses the adaptation problem from that perspective, although its focus has been more on the level of anticipation of such adaptations. Supporting context awareness, associated with adaptation triggered by context changes, has been the focus of many researchers over the years. In this section, I mention a number of such works.

The authors of [14] present common architecture principles of context-aware systems and derive a layered conceptual design framework to explain the different elements common to most context-aware architectures. A layered conceptual framework containing sensors, raw data retrieval, pre-processing of raw data, storage of context information and context-aware applications is presented. Based on different design principles, they introduce various existing context-aware systems focusing on context-aware middleware architecture and frameworks, which enhance the development of context-aware applications. This paper particularly helps in understanding different approaches of context modeling and we find the ontology-based modeling quite suitable for our work.

SOCAM (Service–oriented context-aware middleware) [15] presents an architecture for the building of context-aware mobile services. They propose an ontology-oriented approach to acquire context information from different sources and interpret it. A two-level ontology hierarchy presents common/global concepts in the top level and the bottom level contains domain-specific context information. The middleware supports acquiring and interpreting various contexts and inter-operability between different context-aware systems. In MUSIC and this thesis, we use similar ideas to develop the MUSIC ontology, where the top level concepts can be extended by individual developers to add application-specific concepts.

Ranganathan et al. [16] provide a middleware solution to support context awareness to automated agents, which can be applications, services and/or devices. The middleware simplifies the development of context-aware agents by supporting context sensing and reasoning on context information and thus relieving the agent developers from many details. In perspective of reasoning, they provide rules as well as learning mechanisms. They also allow autonomous, heterogeneous agents to seamlessly interact with one another, through a context written in DAML+OIL [17]. One of the main shortcomings of this approach is that it does not deal with the specialized context characteristics, such as incompleteness, and that its extensibility is limited.

ECORA (Extensible Context-Oriented Reasoning Architecture) [18] is a prototype framework for building context-aware applications, which are designed with a focus on reasoning about context under uncertainty and addressing issues of heterogeneity, scalability, communication and usability. The framework provides an agent-oriented hybrid approach, combining centralized reasoning services with context-aware, reasoning capable mobile software agents. In MUSIC, we are also working on reasoning about uncertain context information.

Henricksen et al. [19] provide a context-aware software engineering framework simplifying the design and implementation facilitating rapid prototyping for context awareness and experimentation support. They present a graphical context modeling technique using CML (Context Modeling Language), which also supports the relational modeling of context information. The architecture contains a number of different layers for context gathering, context reception, context management, context query and adaptation of the context-aware applications. Unlike our work, they adopt a closed world assumption in order to support reasoning about contexts. Thus, the extensibility of their approach is questionable.

Hardian [20] has enriched the middleware developed by Henricksen et al. [19] by providing traceability and control to facilitate user understanding and feedback. This includes selectively exposing various components (context information, preferences, and adaptation rules and logic) to users. The added functionality can be viewed conceptually as an additional layer above the context management component, providing logging and generation of explanations/feedback for users.

Yau et al [21] have proposed the Reconfigurable Context-Sensitive Middleware (RCSM) which is a middleware designed to provide two properties to applications: context awareness and ad-hoc communication. This is done not in an independent way but in a way that allows RCSM to provide another property named context-sensitive ad-hoc communication. RCSM provides an object-based framework for supporting context-sensitive applications similar to middleware standards and prototypes such as CORBA, COM, and TAO for fixed networks.

Thus, RCSM provides application developers with a context-aware Interface Definition Language (CA-IDL) that can be used to specify context requirements, including the types of context that are relevant to the application, the actions to be triggered, and the timing of these actions. Ad hoc communication support is provided by a context-sensitive object request broker (R-ORB). This communicates at runtime with the skeletons produced by the compilation of the IDL interfaces, provides device and service discovery and use a symmetric communication model to allow ad hoc and application-transparent information exchange between a pair of remote objects.

The prototype described by Yau et al. [21] also does not satisfy the heterogeneity requirement, as it supports only C++ applications for the Windows CE platform. However, the IDL compiler could potentially be modified to produce skeletons for a variety of platforms and communication protocols. In addition, the context discovery protocol is not flexible enough to support mobility or component failures, the RCSM authors do not attempt to address scalability, privacy, traceability, or control. In contrast to RCSM, this thesis focuses on mobile applications instead of the communication aspect. Consequently, the adaptation approach is completely different, although it supports the re-configurability to the middleware.

Along with discussing the fundamentals of context awareness, the thesis of Dey [22] provides a Context Toolkit to support the design and implementation of a variety of context-aware applications. The Toolkit can be used as a research test-bed to investigate new problems in context-aware computing such as the situation programming abstraction and dealing with ambiguous context and controlling access to context.

The Toolkit has a number of building blocks, namely, Context widgets, Context Interpreters and Context Aggregation. Context widgets are responsible for collecting information from the environment through the use of software or hardware-based sensors. Context Interpreters abstract raw or low-level context information into richer or higher level forms of information. Context aggregators aid the framework in supporting the delivery of specified context to an application, by collecting related context about an entity that the application is interested in. This is helpful, because an application may be interested in any number of context information about a particular context entity and therefore, a co-ordination among the widgets providing that information is required. The thesis also defines three different categories of context-aware services (services are defined as behaviors of applications): presentation of information to a user, automatic execution of a service and tagging of context information for later retrieval. Services are used by context-aware applications to invoke actions using actuators, and in addition to this, the Toolkit include discoverers that can be used by applications to locate suitable widgets, interpreters, aggregators and services.

An extension to the Context Toolkit [22] was proposed by Newberger et al [23] for providing user control of context-aware systems using an approach called end-user programming. In MUSIC, we minimize user control, while in U-MUSIC, I add the flexibility that a user can choose and add functionalities expected from his application.

The authors of [24] propose a modular context management system (Draco) that is able to collect, transform, reason on and use context information to adapt services. The context manager of Draco is organized around a database and an ontology broker. The service platform is component-based which can dynamically adapt the context management system to changing conditions of applications' requirements and context devices. The objective is to deploy and undeploy on demand functional context management components, such as filtering, history, or transformation. However, the adaptation process is driven by the objective of saving storage space, but does not support the description and the management of context dependencies.

Hydrogen [25] is a peer-to-peer context-aware system that uses the device's local context, i.e. context acquired by local built-in sensors. The Hydrogen framework architecture consists of a number of layers, namely adaptor layer, management layer and application layer. The adaptor layer is responsible to get information from sensors about the physical context, possibly enriches this information with logical context information and delivers it to the management layer. Context information is stored in a context server and the management layer is responsible for providing and retrieving this context information and sharing it with other devices. Context-aware applications that use the context information provided by the underlying layers constitute the application layer.

Due to its limited capabilities a device cannot sense all the context information itself. Hydrogen provides a mechanism to share sensed context with other nearby devices. Context sharing is based on a peer-to-peer connection over LAN, WLAN, or Bluetooth. However, authors do not mention distributing the *aggregated context*, i.e., context originating from two or more devices, which can be exchanged with a newly encountered device in order to learn about context beyond a single hop.

There are works like [26][27][28][29] that offer new opportunities for adapting collaborative applications and services. Muñoz et al. [26], for instance, propose a context-aware instant messaging, which aims at improving the collaboration opportunities for doctors and nurses in a hospital. In Pagalli et al. [27], authors propose a peer-to-peer platform for communication and knowledge exchange in a community of users dynamically created (represented, in this case, by tourists visiting a given city). In Rocha & Endler [28] the authors present the MoCA middleware, a context-aware middleware allowing the development of collaborative applications, such as a location-based shared notes application (electronic post-its) [29].

The works discussed above focus more on context awareness, where adaptation comes as a secondary aspect. Those works have been useful in MUSIC to provide a rich support of context modeling, sensing and reasoning on context information, and thus preparing them for self-adaptive applications. This thesis adopts the MUSIC context middleware which provides a pluggable architecture to plug-in separately developed context sensors and reasoners [89]. In the following we discuss related works that focus more on the adaptation aspect, while using the context information that triggers such adaptation.

The work of Oreizy et al. [30] examines the fundamental role of software architecture in self-adaptive systems in planning, coordinating, monitoring, evaluating, and implementing seamless adaptation. They discuss different concerns in the self-adaptation and propose a general-purpose approach to self-adaptive systems. At the heart of their proposal is the task of adaptation management involving the collection of observations and measurement, evaluation and monitoring, planning of observation and adaptation and deploying change description. They also suggest the support of evolution management of architecture models based on the observations. However, the support of evolution is quite immature.

The CASA (Contract-based Adaptive Software Architecture) framework [31] presented in the Doctoral thesis of Arun Mukhija of the University of Zurich supports dynamic adaptation, where adaptation decisions are defined as application contracts. Such contracts are akin to adaptation rules (see section 2.3.1). In [31], the adaptation techniques are classified according to the level where the adaptation takes place, e.g., dynamic change in lower-level services, dynamic weaving and unweaving of aspects, dynamic re-composition of application components and dynamic change in application attributes. Ideally, an autonomic application in this approach should be able to use any combination of all these adaptation techniques, depending on its adaptation needs. The adaptation policy of every application is defined in a so-called application contract. The application contract is external to the application and is specified using an XML-based language, thereby facilitating changes in the adaptation policy at runtime. Such adaptation contracts are predefined and can be changed manually, though at application design time. Therefore, the automatic reasoning of unanticipated situations can not be supported.

The Rainbow adaptation framework [32] addresses self-adaptation by introducing the utility concept and minimizing the use of rule-based approach for adaptation reasoning. Thus it helps getting rid of the high-level human adaptation decision. The work also aids reasoning on quality dimensions by reducing infinite number of states that are possible based on different quality dimensions to a finite number of states. For example, the infinite range of response times may be reduced to three states, namely, low, medium and high. They treat stakeholder preferences over the objectives to be static once defined in the adaptation framework and therefore, it does not support changing requirements dynamically at runtime. The move towards the use of utilities, however, is a step forward towards the dynamicity of the adaptation reasoning policy.

In [33], the self-awareness and adaptability of the environment is addressed at two levels. At the higher level, the infrastructure monitors the availability and performance of whole components and of the communication infrastructure, evaluating possible alternatives for supporting a user task when the requirements for such a task are not met by the current configuration. At the lower level, system components themselves are endowed with the ability to adjust their operation following the variation of available resources like CPU, bandwidth, battery charge, etc.

The architectural framework consists of four component types: Task Manager, Context Observer, Environment Manager and Supplier. The Task Manager, called Prism, embodies the concept of personal Aura. The Context Observer provides information on the physical context and reports relevant events in the physical context back to Prism and the Environment Manager. The Environment Manager embodies the gateway to the environment; and Suppliers provide the abstract services that tasks are composed of: text editing, video playing, etc.

Bruneton et al. [34] present the Fractal component model which supports the definition of structural composition based on containment and binding relationship between components. They also present the Fractal framework, which is a projection of the fractal component model in Java. It offers both static and dynamic configuration of components so that all or some or none of the components may be reconfigurable.

SATIN [35] is a lightweight component model, which represents a mobile system as a set of interoperable local components. The model supports reconfiguration, by offering code migration services. SATIN uses the software component paradigm and supports component migration to allow logical migration. Software components can be considered as a lower level of abstraction that can be used to represent and model software agents.

Carisma [36] uses the reflection mechanism – ability of a program to reason about and alter its own behavior – to observe and reconfigure self-adaptive systems. DART [37] is a platform dedicated to the development of adaptive applications. It is an early example of a system that explicitly uses the concept of an action-based adaptation policy. DART Policies are associated to components and managed and coordinated by the DART Manager component. This coordination includes the resolution of conflict and incoherencies between the set of policies present in the system. To handle that, the policies are organized in three abstraction levels: system, middleware and application. The policies within one group (or level) are also organized with different priorities. As an extension to DART, Safran [38] and Chisel [39] propose self-adaptive component models by adding and managing adaptation policies as a separated concern from the functionalities of applications. For example, Safran extends the Fractal component model [34] by associating rule-based policies to Fractal components within the membrane as a new kind of component controller for self-adaptivity. Even if these systems allow crafting and modifying policies dynamically, they do not address the problem of policy management in presence of many applications with different policies in general and policies distribution in particular. In MUSIC and this thesis, we make the adaptation reasoning based on a single policy (utility-based reasoning policy) and it simplifies the task of adaptation reasoning in the case of multiple applications. We also support distributed reasoners in taking the adaptation decisions.

3.3 Semi-anticipated and Unanticipated Adaptation

In recent years, the integration of services within the component framework to build adaptive applications has been in research focus. As described in section 1.1.2, adaptation by such integration of services can most be considered as semi-anticipated adaptation. In MUSIC, this has also been one of the main topics of improvement compared to MADAM. We have already provided modeling [95] [96] and middleware [96] support for context-aware self-adaptive applications in ubiquitous and service oriented environments. We have provided modeling concepts to describe services and their QoS properties with a harmonized view on context and service properties, bridging the syntactical and semantic differences through an ontology. The middleware supports plugging in services and components interchangeably in building the application configuration. The support includes the discovery of services based on the semantic modeling of service needs, negotiation, provisioning and monitoring of service level agreements, and integration of services in the adaptation reasoning process.

During our development of service support, we have studied a number of important solutions in the related field. For example, Adaptive Service Grids (ASG) is an open initiative that enables the dynamic binding of services in adaptive service environments [97]. ASG expresses service request through a semantic description of the functionality of the service. The platform then tries to find a service that matches the service request either perfectly or imperfectly (as perfectly as possible). An agreement with a particular service is set up either by a negotiation mechanism (when supported) or simply based on the static properties of the service. The approach is similar to ours, as it uses a semantic description of the desired functionality utilizing a domain ontology to discover services. However, in contrast to our approach the planning is not QoS-driven. Therefore, support for QoS specification and the mediation of QoS properties only play a secondary role in ASG.

VieDAME [98] proposes a monitoring system that observes the efficiency of BPEL (Business Process Execution Language) processes and performs service replacement automatically upon performance degradation. Like ASG, VieDAME also focuses only on the planning per request of service compositions with regards to the properties defined in the semantic service request. Thus, neither of these approaches support a uniform planning of both components and services so that services and components may be used interchangeably.

Our approach for the modeling support shares a lot of concepts with the work done by Bleul et al. [99]. In particular, we follow nearly the same approach to the modeling of QoS dimensions, Service Level Requirements and Service Level Packages and to the integration of the resulting specifications into the OWL-S description of a service. They also focus on quality-aware service descriptions. However, they do not address the context issue at all, whereas we align the modeling of QoS properties with the modeling of context information and context properties.

Flores-Cortés et al. [41] present a dynamically reconfigurable multi-personality middleware that supports the discovery of services advertised on multiple platforms and achieves interoperability between heterogeneous discovery protocols. The design of the middleware is based on discovery of the common features of the protocols. They analyze a cross-section sample of protocols and identify a common set of architectural elements. The core elements of the service discovery architecture consist of six component types that provide the functionalities common to most service discovery protocols. An advertiser component helps advertising services, a request component passes service needs, a reply component ensures when a matched service is found, a cache component provides temporal storage or service directories, a policies component manages service usage and interaction policies, and a network component enables components to transmit and receive messages. Like them, we also support the discovery of services using a number of different protocols; however, we deploy the support for each protocol separately.

Menasce and Dubey [100] propose a QoS brokering approach in SOA. Consumers request services from a QoS broker, which selects a service provider that maximizes the consumer's utility function with regards to its cost constraint. The approach assumes that service providers register with the broker by providing service demands for each of the resources used by the provided services as well as cost functions for each service. The QoS broker uses analytic queuing models to predict the QoS values of the various services that could be selected under varying workload conditions. This approach is of interest both from the viewpoint of a consumer and a provider. While the client is relieved from performing service discovery and negotiation, the provider is given support for QoS management. This approach, however, requires the client device to be able to access the broker, which might not be possible in ubiquitous environments. Our approach differs in that we consider the offered properties as alternatives to determine the best application configuration and allow the client to adapt to the service landscape.

In MUSIC, we have integrated the SOA support in the conceptual model for component-based applications. This facilitates the integration of services as alternatives to components in the application composition in a way that in a particular application some functionality may be realized by MUSIC components, while some others may be provided by third party services. Most of the research projects supporting context awareness and self-adaptation, however, focus more on the service-centric approach. For example, the conceptual models of both SeCSE (http://secse.eng.it) and PLASTIC (http://www.ist-plastic.org) focus on service-oriented systems. Inspired by the SeCSE model, the PLASTIC model extends it by introducing new concepts, such as context, location, and service level agreements. The MUSIC and the PLASTIC model have in common that both combine SOA and component-based software development. However, the MUSIC conceptual model uses a component-centric approach, while the PLASTIC model uses a service-centric approach.

The concept of the unanticipated adaptation can be viewed as a special and relatively unexplored feature of context awareness and self-adaptation. In [40], Manuel Oriol, in his PhD thesis, has discussed dynamic and unanticipated software evolution. Such evolution may occur at compile time, load time or runtime of the software. Compile time changes are defined as static evolution and dynamic evolution refers to runtime changes. Unanticipated evolution is defined as changes that can not be foreseen by the programmer.

McKinley et al. [101] use the term unanticipated adaptation to enable CORBA applications to adapt to unanticipated changes in their functional requirements or their execution environments. The approach is claimed to be suitable for three different types of applications. Dependable applications may be adapted to operate without interruption even when faults are occurring at application runtime. Embedded applications may add or remove adaptive code at runtime. Moreover, some legacy applications may require that the code can not be modified at runtime and thus adaptation code can be woven keeping the core code unchanged, while the application is running. They use a rule-based interceptor, which can weave new adaptive code dynamically at runtime based on a set of rules. Such rules can also be loaded at runtime. However, their definition of unanticipated adaptation is quite limited. Although adaptation code can be loaded dynamically at runtime; the actual availability of such code is not an automatic process. In contrast to [101] the adaptation reasoning approach of this thesis does not use adaptation rules; rather the application configuration is evaluated using a utility-based approach. This is helpful, because the application developers do not need to think about explicit rules; instead they can aim at maximizing the utility for the application. Utility functions for individual components are specified at design time; but the overall utility of an application is automatically adjusted at runtime, based on the available components and services. The use of the utility function removes any possibility of the existence of conflicting rules. The approach presented in this thesis is very useful especially when components are developed by different developers in an unanticipated manner.

A number of works have tried to support adapting software code to face unanticipated situation. For example, Pukall et al. [102] apply the Java HotSwap and object wrapping techniques to integrate adaptation possibility of Java code of a program without making it unavailable or stopping it. However, their consideration of unanticipation is limited only until deployment time. Conroy [103] also deals with unanticipated adaptation of software systems, where semantic representations are provided to avoid incompatible interfaces, when runtime changes triggers to use new components. Such approaches mainly focus on updating code, whereas in my approach, I use different adaptation mechanisms to find the best choice from all the possible realizations of a particular application. Of course, I explicitly address mobile applications, where the changing context and the availability and unavailability of new services and components are the triggers to adaptation.

The work of Mügge et al. [42] aims at minimizing anticipation of adaptation at design time by applying aspects and thus introducing details about application variants at runtime. They also introduce the *functionality* concept, which has been useful in defining the *functionality* concept in this thesis.

Cremene et al. [43] adopt similar meaning of the unanticipated adaptation concept. They point out the difference between anticipated and unanticipated work. However, their contribution is very limited. In their work the adaptation control is based on predefined service-specific rules and strategies. These solutions will not work correctly in a context that was not taken into account by predefined rules and strategies even if large context diversity was considered. I extend the notion of *unanticipated adaptation* by adding the possibility of adaptation based on the runtime situations: the environment, resources and the available components, services and plans to realize the functionalities of an application. It remains unanticipated even when the application is running. The application architecture is decided only during the adaptation process. Thus, this information becomes anticipated only during the 'time' of adaptation.

ECORA [18] presents a reasoning approach, especially to recognize situations under uncertainty applying a sensor data fusion technique to consider factors like inaccuracy of sensed information, reliability of sensors and importance of information for inferring particular situations. They use a Context space model to define regions, for example, in numerical form an accepted region would describe a domain of permitted real values for an attribute, such as the region of values of body temperature between 36.2 and 36.9°C, representing the range of temperature values of a "healthy person". Then using the regions along with the uncertainty of information, they calculate a utility to indicate how well a situation matches to the context space.

In [88], Vanrompay et al. discuss the adaptation reasoning technique with uncertain information. However, that topic deals with providing adaptation solution, when there is some ambiguity in the context information and therefore, the focus is on the context information and not on the unanticipated adaptation problem as it is addressed in this thesis.

Part II Supporting Unanticipated Adaptation

4 Development of Concepts

The adaptation of an application, in general, is meant to adjust the quality of service based on the context and resource characteristics, while still supporting the core functionalities of the application. Moreover, some functionalities may be optional and they can be added or discarded to the application's core set of functionalities, based on users' choices and the availability of new components and services.

Runtime adaptations impose the challenge that the components and services are most often not available at design time and therefore, the middleware can not pre-estimate the possible application configuration. In that process, the application architecture itself also evolves at runtime based on the available components, services and their meta-information. A number of different variants of the application can be possible and the middleware helps to select the variant that best fits the runtime context. The support for unanticipated adaptation enhances the challenge, because the application developer may not have any idea about the components provided by other U-MUSIC developers. Therefore, such components must be detected and used in constructing the application configuration at runtime. This necessitates a well-defined data structure for concepts that support such integration of components as well as discovered services to create application variants at runtime.

4.1 Conceptual Meta-model

The conceptual meta-model defines different concepts needed to support the unanticipated adaptation in U-MUSIC and the relationship among them, as depicted in Figure 1.

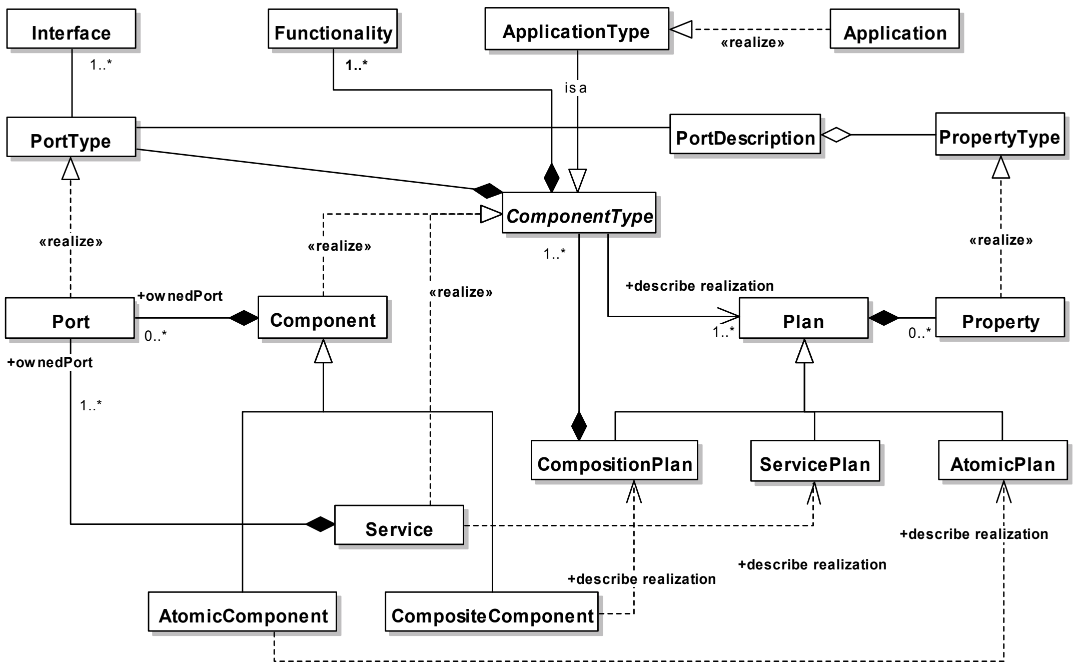


Figure 1: Conceptual meta-model of U-MUSIC

The MUSIC conceptual meta-model is taken as the basis; however, it is updated appropriately in order to add the support for the unanticipated adaptation. A description of different concepts and terms used in the meta-model, along with their relationship with each other, is presented below.

Component

The component concept is based on the definition provided by Szyperski,

> "*a component (or software component) is a unit of composition with contractually specified interfaces and explicit dependencies where dependencies are specified by stating the required interfaces and the acceptable execution platform (s).*" *[44]*

In addition to Szyperski's definition (and not excluded by this definition), we consider that components may be structured (i.e. consist of other components).

Components as compositional unit can interact with each other. Ports are defined as interaction points. A component can own any number of ports.

Atomic component

A component which can not be structured in a more granular way; i.e., it does not consist of other components, is termed as an atomic component.

Composite component

A component that has internal structures and is therefore composed of any number of other components is termed as a composite component.

Service

In the context of enterprise architecture, service orientation and service-oriented architecture, the term service refers to a discretely defined set of contiguous and autonomous business or technical functionality. The OASIS (Organization for the Advancement of Structured Information Standards) defines service as,

> "*a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description.*" *[45]*

Like MUSIC, we basically comply with these definitions but are primarily concerned with services which are implemented by software components and provided to software components, and which are described in terms of software interfaces in standardized ways. However we also often use the term service to denote the software component implementing a service. Unlike MUSIC, we do not consider the hosted services by a MUSIC node; this work considers only services provided by non-U-MUSIC providers.

Functionality

Functionality is defined as *what* a component would perform. Any functionality may be further realized with part-functionalities. By part-functionality we refer to a functionality that may be realized by a component, in order to contribute to the overall functionalities provided by the application or a composite component. Moreover, a single functionality of a particular component (atomic or composite) may be the outcome of several other functionalities. In this later case, individual part functionalities are not necessarily realized by different components. For example, a ‘ticketing’ functionality may consist of the part functionalities like ‘ticket selection’, ‘ticket buying’ and ‘ticket verification’ functionalities. It is possible that a single component may realize all these functionalities, while individual part functionalities may be realized by individual components and/or services.

Component type

A component type defines a set of functionalities offered to the user or other component types. A component type has a set of port types that are the (abstract) interaction points to communicate with other component types. For a particular component type, some functionalities may be defined as core functionalities, while some other may be optional. Core functionalities must always be realized by the component and/or service that realizes the component type. Also, some functionalities may be added at runtime by the user. We consider such functionalities as ad-hoc functionalities.

Application type

An application type is a specialization of a component type and thus it is a collection of functionalities expected from the application. In general, such functionalities may be quite abstract or concrete as needed. For example, an application that helps a tourist may be characterized by a single functionality ‘AssistTraveller’, which itself abstracts away a number of more concrete- (or, part-) functionalities like ‘PlanItinerary’, ‘TakeImage’, ‘CalculateRoute’ etc.

Port type

Port types are the (abstract) interaction points to communicate with other component types. This differs from the MUSIC concept, because in MUSIC port types are meant to characterize the component type. A service type in MUSIC corresponds to a functionality that is required or provided through interfaces and a port realizing this service type. In the case of expressing the need for external services, we keep the MUSIC notion unchanged. Therefore, port types are connected with a port description (service description) to express the information needed for searching and using a service.

Port description (service description)

Port descriptions describe the information needed to search for a service proactively. They also provide the information required for using that service. In this document, we use the terms port description, service description and service info synonymously.

Application

An application realizes the set of functionalities of an application type. It is considered as a specialization of a component, because a component, in general, is considered as a realization of a component type. In the compositional adaptation as addressed in this work, an application is viewed as a composition of components.

Plan

A plan describes a component, thus providing the information needed to obtain a particular realization of a component type. Such information includes the QoS properties, the resource requirements, a description of the composition (for composite realization plan), a reference to a component (for atomic realization plan), a utility function etc. A component type can have any number of different realizations, both anticipated and unanticipated at design-time, each of which provides the functionalities expected from it. Some functionalities may be mandatory (core functionality), while some of them may be optional. Moreover, some functionalities may apply only for certain situations (ad-hoc) or may be added at runtime as per the wish of the user. A plan (precisely, the component that the plan describes) can be considered as a realization of a particular component type, when it provides the core and ad-hoc functionalities along with a subset of the optional functionalities of the type. Thus, realizations of a particular type may differ in the sets of functionalities as well as in the quality of service characteristics that are specified through properties with regard to context and resources.

A property specifies the required or the provided value for a particular property type and so it is considered as a realization of that property type. An example of a property type can be ‘network-bandwidth’, which indicates that a component type depends on network-bandwidth, while a property can be specified as ‘network-bandwidth > 10 kbps’. In addition to such constant values, a property can be dependent on other properties and therefore, it can be specified as a function of other properties. Such functions are called property predictors.

Components and services are the entities that provide the functionalities. Thus, the functionalities of a component type can be realized by a set of communicating components and services. In this meta-model, we consider components as software entities that are instantiated at runtime to realize the functionalities, while services are considered as entities external to the middleware domain that provide those functionalities without requiring to instantiate them or needing to know their internal details.

Atomic plan

An atomic plan describes the realization of a component type, when all the functionalities of the component type can be realized through a single component that does not need to be decomposed.

Composition plan

The functionalities of a component type can be further sub-grouped so that such groups may or may not depend on each other. This also includes the possibility that a functionality can be achieved by the collaboration of a number of other (new) functionalities. In that case, the realization can be provided through a set of possibly communicating components. Such realizations are described through composition plans.

Service plan

Service plans are used to facilitate the integration of external services as the means to realize a component type. They include the information required to use a service. Such information consists of the service name, service interface, host name, service port, service technology, service URL, service classification and a set of properties. Service plans are created at runtime, when a usable service is discovered.

Bundle

Although the Bundle concept is not a part of the conceptual meta-model, because it is not directly used to create the application variability model, it is useful to understand the runtime adaptation process involving a number of different nodes. A bundle is a deployable unit for the deployment of individual components, their plans as well as the definition of component types and application types. Such a bundle can contain a number of different U-MUSIC artifacts like component types, application types and plans. In MUSIC, as well as in this work, adopting the term 'Bundle' is influenced by the corresponding term in the OSGi community [75]. Moreover, MUSIC bundles are managed using the OSGi framework. However, it must be noted that neither a MUSIC bundle nor a U-MUSIC bundle implements the OSGi Bundle interface.

The U-MUSIC conceptual meta-model is an updated version of the MUSIC conceptual meta-model in order to support the unanticipated adaptation. The main updates are as follows:

- The concept of functionality is added to define component type. Like MUSIC, a component type has a type name; but the type is characterized by a set of functionalities, instead of a set of port types.
- The port type concept is used with the simple meaning of 'interaction point'. Therefore, unlike MUSIC, it is not used to characterize the component. However, this is not the case when a component type is foreseen to be realized by an external service. In that case, we keep the MUSIC concept unchanged so that a port type is used as a synonym to service type.
- The role concept is discarded. In comparison to MUSIC, this limits the definition of component type such that a component type does not have a number of different roles; but it simplifies the conceptual model. Eventually, this also limits the variability introduced by roles. However, such limitations are overcome by introducing different kinds of functionalities; for example, the realization of a component type must always provide the core functionalities, while optional and ad hoc functionalities introduce variability of the component type.

4.2 Creating Application Variants

The basic idea of adaptation in our approach is to choose a realization of the application type from a set of possible variants, based on its utility for that particular context and resource condition. These variants are obtained by dynamically creating a variability model of the application based on the available component types, services and realization plans at the time of the adaptation reasoning. Thus the application supports a combination of the compositional and the parameterized adaptation; i.e., a new composition of components and/or services can be chosen, with the possibility that some components may be instantiated with a modified set of properties, based on some parameter values.

In a distributed environment, there can be any number of nodes in the adaptation domain during the time of the adaptation. Application bundles can be deployed at any time on different nodes that are reachable or not due to the changes in the network, or the movement of the user. Thus components, component types and plans can appear and disappear in an unanticipated way and they are discovered at runtime. When a new bundle is deployed, the U-MUSIC middleware collects the information about the deployed application types, component types, plans, components and context sensors. The middleware establishes the correspondences between plans and component types through storing the information in service repositories. When a node leaves the adaptation domain the bundles deployed on them are removed from the repository, always keeping an up-to-date trace of all the available component types and plans.

Besides, the service discovery protocols integrated in the middleware advertise newly discovered services, based on the service need specified in the component type model, to a plan broker. Plans for these services and from known service repositories are generated from service level descriptions (if available) or using some default value for the expected properties so that they are available when the planner initiates an adaptation at a later time. Of course, plans are discarded when services become unavailable to the middleware and an adaptation process is triggered if a service described by the discarded plan is currently in use. A service might offer a predefined set of service levels. Then, for each of those sets a separate plan is generated by the plan broker. Thus, the planning framework is able to take service levels into account when planning the adaptation.

The creation of application variants using the plans and component types available at runtime can be explained using Figure 2.

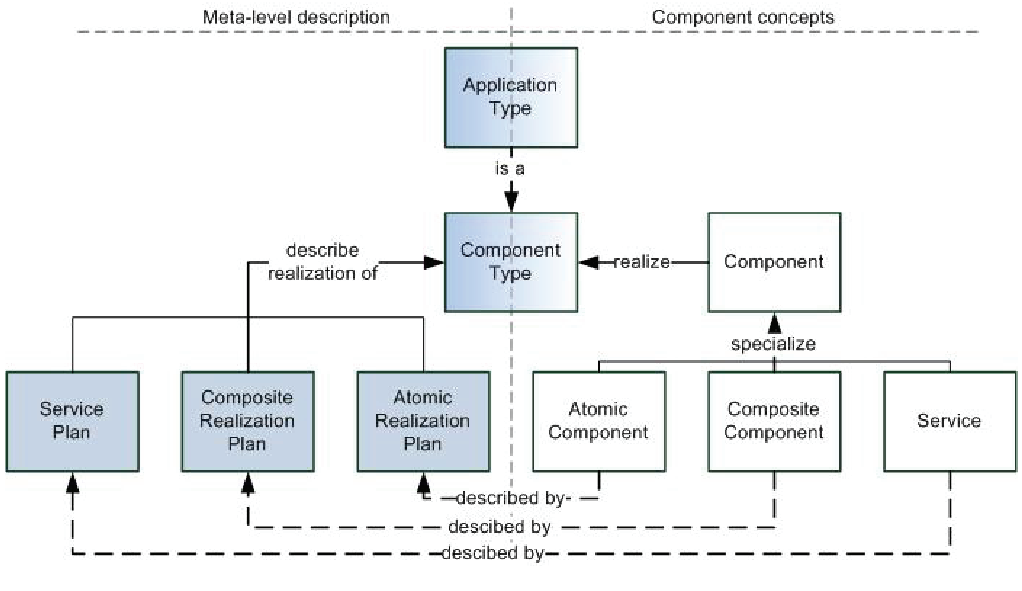


Figure 2: Creating application variants

Figure 2 illustrates that an application type is viewed as a component type that can have different realizations. The details and the QoS properties of a certain realization are described using plans. Corresponding to the atomic and composite component types, there are two types of plans: atomic realization and composite realization. An atomic realization plan describes an atomic component and contains just a reference to the class or the data structure that realizes the component. A composite realization plan describes the internal structure of a composite component by specifying the involved component types and the connections between them.

Variation is obtained by collecting information from the component type and plan repositories about the set of possible realizations of a component type using plans. In order to create a possible variant, one of the plans of a component type is selected. If the plan is a composite realization plan, it describes a collaboration structure consisting of further component types, which in turn are described by plans. Now we proceed by recursively selecting one realizing plan for every involved component type. The recursion stops if an atomic realization plan is chosen. Therefore, by resolving the variation points we create application variants that correspond to a certain composition of components depending on the plans that are chosen for each of the component types.

With the service-based adaptation a part-functionality may be provided through a dynamically discoverable and accessible service. Thus, compositional adaptation is extended by taking a service as a possible realization of a component type. To do so, the QoS properties, interfaces and binding information have to be included in a corresponding plan. In the composition for creating application variants, services and corresponding service plans are treated like atomic components and atomic realization plans. Therefore, a service plan would also indicate the end of the recursion for that branch of the variability tree.

5 Runtime Adaptation Mechanism

Runtime adaptation refers to adapting the application without having to stop it, when the context changes, the availability of resources vary, and nodes and services become available or unavailable without any prior notice. The runtime adaptation mechanism incorporates a number of different tasks; e.g., the deployment of application bundles to the middleware, the construction of the application variability model from the artifacts of the deployed bundles, reasoning about the available application variants and reconfiguring the variant with the highest utility.

In chapter 4 we have described the basic concepts that support such adaptation through creating application variants and reasoning about the adaptation decision. In this section, we will use that information as the baseline and show concretely how those concepts are used at runtime to obtain the unanticipated adaptation. However, we adopt the reconfiguration mechanism from the MUSIC project and therefore, it is not discussed in this document.

5.1 Deployment of Bundles

A bundle is a deployable unit for the deployment of individual components, their plans as well as the definition of component types and application types. Such a bundle can contain a number of different U-MUSIC artifacts:

- zero or more component types
- zero or more application types
- zero or more plans

It is evident that the content of a bundle is quite flexible and this facilitates the development and deployment of types and plans independently. This is particularly needed for the unanticipated adaptation, when a particular developer may not have any idea of what will be provided by other developers.

5.2 Constructing the Application Variability Model

In a ubiquitous computing environment, devices may appear and disappear without any prior notice. Moreover, a user can choose anytime to deploy a new bundle or remove a deployed bundle. This presents a highly dynamic environment, especially for applications that depend also on services and components provided by others than the user himself.

5.2.1 Runtime Matching of Plans and Types

When a new bundle appears within the adaptation domain, it is registered along with the bundle artifacts. At this phase a correspondence between the application type or component type and the set of plans that can be used to realize them is established. Such correspondences are created using the meta-information associated with a type definition and that of the plans. For this work, we suggest the matching of functionalities along with the interfaces of types and plans. If a plan realizes all the mandatory functionalities of a particular type, it is considered as a realizing plan for that type, given that there is no mismatch of interfaces that the type defines and what the component corresponding to the plan implements. Such a matching technique, as adopted in U-MUSIC, is different from and advantageous to the MUSIC solution in the sense that a particular plan can be used to realize a number of different types. Moreover, an imprecise matching is possible, especially when components from a different developer are used to realize the component types defined by a particular developer. In MUSIC there is a static dependency between a plan and a type and therefore, a particular plan can be used to realize a single type only.

During the registration of a plan in the plan repository, the existing types are checked, and then the plan is added in the sets of realizing plans for the matching types. In a similar way, when a type is registered in the application type or component type repository, the already registered plans are checked and the matching set of plans is added as its realizing plans. When a particular bundle is removed, due to the unavailability of the device or explicitly by the user, all its artifacts are also unregistered from the repositories. To do this, corresponding type and plan repositories are updated removing the unregistered types and plans of the leaving bundle.

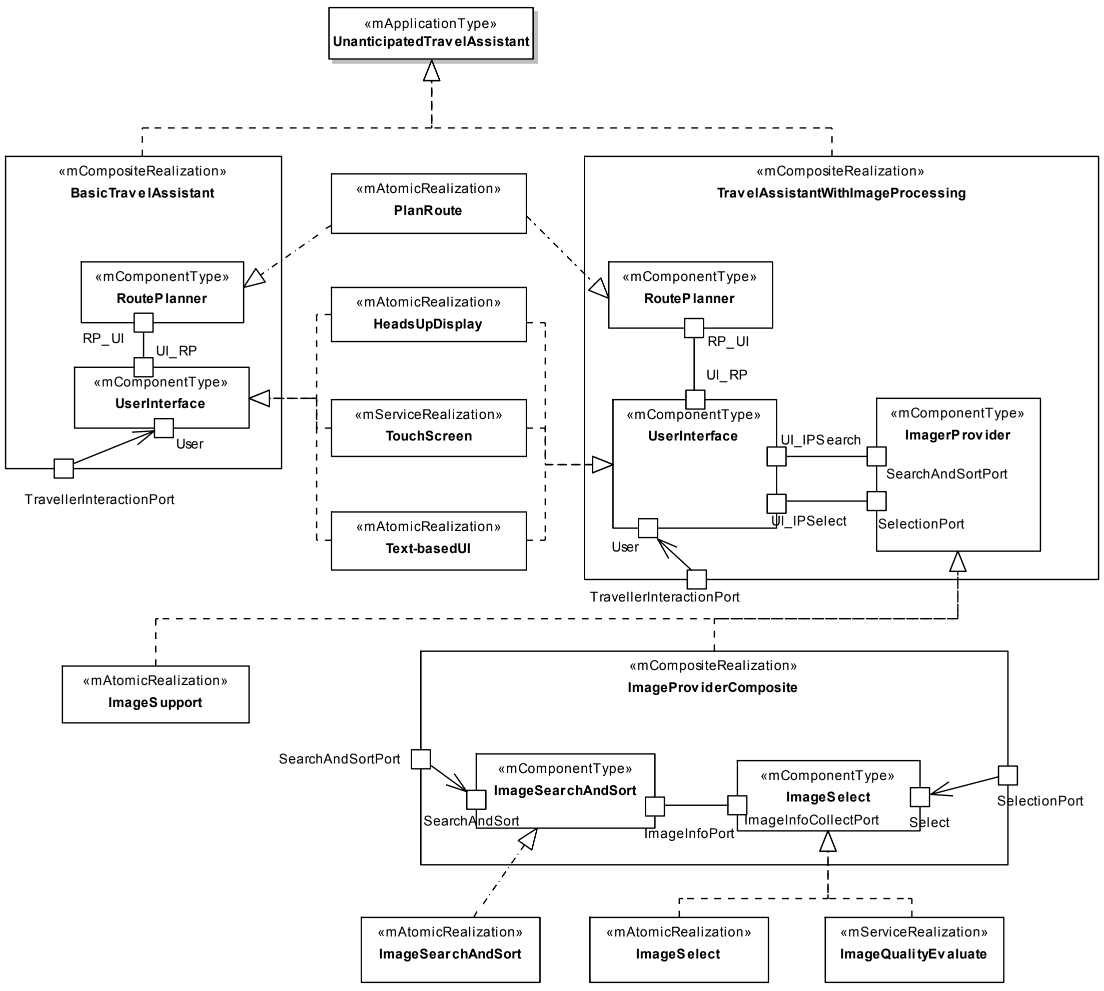


Figure 3: Application variability architecture created at runtime

Another task of the middleware, as adopted from MUSIC, is to discover services that realize different component types, marked as realizable through services. Discovered services are treated differently compared to the discovery/deployment of new types and plans. Corresponding to each discovered service a service plan is created based on the service description, and then the plan is registered in the plan repository. This process differs from the matching of component plans in that respect that such plans describe a U-MUSIC component and therefore, does not require steps that are needed to use a service, e.g., SLA negotiation, creation of service proxies or generation of service plans. Thus, at runtime the application adaptation model corresponds to a variability hierarchy containing component types and their realization plans. This variability model can be used to create application variants by resolving all the variation points.

In order to illustrate how the application variability model is created at runtime from the available application types, component types, services and plans, let us consider the UnanticipatedTravelAssistant application from the scenario described in section 1.2. For a particular point in time, a number of different types and plans are available. These types are matched with the available plans and services and a variability model is created for the application. Such an application variability model may look like that of Figure 3.

At a particular situation, the UnanticipatedTravelAssistant application type has two matching plans: BasicTravelAssistant and TravelAssistantWithImageProcessing. Both of them realize the high-level functionality 'AssistTraveller'[5] of the application; however, the first plan supports only route planning and user interface, while the second one also supports image processing.

The RoutePlanner component type has a single atomic realization plan, namely PlanRoute. However, there are three different realizations of UserInterface available, with a touch screen support provided by a user interface appearing as a service (e.g., from the Petrol station of Scene 2 from section 1.2). For the ImageProvider component type, two different realization plans are available, where the ImageSupport plan indicates an atomic realization providing the image processing functionalities by a single component and the ImageProviderComposite plan indicates a composite realization plan providing the functionalities through a composition of components. The composition in the plan is described at the type level for the sake of obtaining variability. The ImageSearchAndSort component type of this composition has a single realization plan, while the ImageSelect component type can be realized by an atomic component as well as by a third party service.

The variability model for a particular application can be created by matching between component/application types and realization plans. The creation of application variants is done by choosing among alternative realizations for each of the types. For example, let us consider that through the adaptation reasoning process the BasicTravelAssistant plan is chosen, ahead of the TravelAssistantWithImageProcessing plan. This plan has a composition comprising two component types. Let us assume that the Text-basedUI user interface is chosen among the three alternatives, along with the single option of the

[5] Details of the functionalities are presented in Table 2 later in this document. The modeling is presented in section 7.2.3.

PlanRoute plan for realizing the RoutePlanner component type. Therefore, a composition of the components corresponding to these two plans will comprise a variant of the application. The selection of the TravelAssistantWithImageProcessing realization plan would require realizing components for all three component types, while the ImageProvider component type may be realized by a single component (corresponding to the atomic plan 'ImageSupport') or again a composition of components/services (the 'ImageProviderComposite' plan). Resolving the variation points, we may create an application variant comprising components and/or services corresponding to the plans PlanRoute + Text-basedUI + ImageSearchAndSupport + ImageQualityEvaluate. In this variant, the service corresponding to the ImageQualityEvaluate plan is used to realize the ImageSelect component type. Such selections depend on the utility values as determined by the adaptation reasoning mechanism, described in section 5.3.

5.2.2 Creation of a Stable Variability Model

The creation of the application variability model by matching component types with available plans and services introduce a number of challenges. In the following we discuss these challenges with possible solutions:

- When resolving the variation points, it may be possible that a plan which is already included for a component type appears again as a realization option for another component type in the same line of the type-plan hierarchy. This will result in an endless loop and therefore, the variation points will never be completely resolved. In order to solve this problem, an already encountered plan will be discarded when it again appears in the same hierarchy, when variation points are resolved.
- It can be possible that no realization plan is available at all for a particular component type. Now, when such types appear in the composition of a plan, obviously, that plan will also be useless, because it can not be realized completely. When such cases are detected, the composition plan is immediately discarded from further consideration, while creating application variants by resolving variation points.
- It might be possible that more than one component type in a composition can be realized by a particular plan. For example, a single plan may provide a number of functionalities, while parts of those functionality requirements are defined in individual types. In such cases, following the variability model would require duplicating the component in the composition, while a single component would suffice. However, this is only an improvement issue and it will not hinder the adaptation problem. Currently, we do not support that improvement.

5.2.3 Dynamicity of the Variability Model

The idea of supporting the unanticipated adaptation in this work requires a dynamic model for the variability of the application. The application architecture is component-based, while services may replace components as well. This arises from the fact that in a ubiquitous computing environment service providers may provide services to be integrated within and used by many types of applications, irrespective of their development methods. Thus the variability model is quite flexible to accommodate U-MUSIC developers as well as the huge number of service providers who possibly have no idea about U-MUSIC. The dynamic creation of the presented application variability model as well as the application composition at runtime can be ensured considering the following facts:

- The number of components in the composition of the application is not fixed; it may change based on the plans used for realizing the application. Moreover, discovered services may be added to replace some of the existing components in the application configuration.
- The number of functionalities realized by the application is flexible. Some of the functionalities may be considered as core functionalities, while some other may depend on the requirements at runtime and the availability of realizing components and services. Moreover, new functionalities may be added at runtime by the user on demand. For example, a user may be allowed to add ad-hoc functionalities through a user interface.
- Service plans may be created dynamically at runtime, based upon the service levels of the discovered services along with the meta-information provided at design time. Thus, for a particular service different realization plans may be created.
- Unlike MUSIC, a particular plan is no longer bound to a particular type. Therefore, a single plan may be used to realize different component types; this may be particularly useful when only a subset of the functionalities realized by a plan is required to realize the component type.

The support for services in the MUSIC middleware is still improving in terms of the number of discovery protocols and communication protocols. In MUSIC, we are working on adding as much flexibility as possible so that a service discovered by a particular discovery protocol may not be limited by particular communication protocols. Such flexibilities will add even more dynamicity in the creation of the application architecture.

5.3 Adaptation Reasoning

The middleware provides the runtime support of adapting the application through context sensing, adaptation reasoning and the reconfiguration process. Among these three steps, the adaptation reasoning process is the most vulnerable step to the scalability problem [57].

The number of application variants increases rapidly with the number of component types participating in a composition. Though this increase is not prominent for a very simple variability model like that presented in Figure 3, it becomes an issue of great concern pretty quickly when we think of a slightly more complicated model. Mathematically, a composition plan having c different component types, where each of the types has n different atomic plans, will have n^c variants for this particular composition plan alone. Thus the number of application variants increases rapidly with the increase in the number of component types and the number of realization plans for each of the types.

Selecting the best-fit variant through the calculation of the utility for each of such variants, which may result in a combinatorial explosion, is a computation-intensive task. It often fails to provide a solution within a reasonable time frame (e.g., a few seconds);

the effect becomes more prominent for resource-scarce mobile devices. Thus, approaches that take each application variant separately into account to calculate utility may suffer from drastic performance degradation in the adaptation reasoning with the discovery of even a few new realization options. In the case of the unanticipated adaptation, the problem introduces even more 'uncertainty', because the variability model as well as the number of application variants can not be foreseen. Therefore, its influence on the adaptation reasoning time should be minimized.

With that aim in mind we have developed a new adaptation reasoning approach, looking at the problem from a different perspective, compared to the MUSIC solution, to make it stable against such combinatorial explosions. The reasoning time depends linearly on the number of plans. It is no longer influenced by the number of application variants, which is roughly a product of number of plans for individual component types in a composition. We first present the reasoning approach and afterwards, we explain the integration of related aspects like checking resource limits and applying architectural constraints [58] along with reacting on context changes.

5.3.1 Basic Reasoning Approach

In this work, the term 'utility' is introduced as a measure of how well a software system fits a given context. From this perspective, a component or a service has a certain utility for a particular context based on its QoS properties. The utility can be evaluated at runtime by a developer defined utility function. An application is composed of components and services. The utility may depend on the fitness of individual components and services. Moreover, other properties, e.g., the communication among different components, distribution of components on different nodes etc. may influence the fitness of a particular component composition. For example, the existence of a cheaper network may enhance the popularity of a particular composition involving components on different nodes.

Assumptions:

Based on the above discussion, we can make the following assumptions as the baseline for the reasoning approach:

1. The utility of an atomic component or service corresponding to its atomic realization plan or service plan can be calculated based on the QoS properties.
2. The utility of a composition depends on the (part) utilities of the constituent component types and the properties that are independent of individual components.
3. In a composition, part utilities influence the composition-utility positively.
4. The utility functions should be designed in a way that the utilities of alternative realizations for every component type are comparable.

Therefore, it is assumed that the utility of the application can be derived from the utility of its constituent components as well as other properties unrelated to a particular component. For example, in the application variability model of Figure 3, each atomic realization plan and service plan has a set of QoS property specifications that indicates the quality of service characteristics required from the context and resources for the component or service to be usable. A utility function takes those requirements into account and computes a utility for the realizing plan by comparing them with the context and resource characteristics of the runtime environment. For an atomic or composition plan, such functions are provided by the developers, while for service plans, such functions may be either provided explicitly in the service description or created using the property information provided in the service description. Moreover, according to the assumption 3 above, an increased utility of a constituent component will contribute to an increased utility for the overall composition or application. On the other hand, when the utility of a constituent component reduces the overall utility of the composition, or when the utilities of a component influences the utility of another component in the composition, then the assumptions become invalid. Since the approach selects the realization that provides the highest utility, the utility values must be comparable; for example, all utilities may be expressed in percentage or they may have always a certain range like 0.0 to 1.0. This fact is expressed by the assumption 4.

Mathematical formulation:

A composite realization plan contains a composition of component types. Let us consider that $\mathbf{CT} = \{CT_1, CT_2, \ldots, CT_n\}$ is the set of component types that is involved in a composition **C**. For all $CT_i \in \mathbf{CT}$, there exist sets

$\mathbf{A} = \{a_1, a_2..., a_p\}$,

$\mathbf{B} = \{b_1, b_2..., b_q\}$,

…,

$N = \{n_1, n_2..., n_z\}$

where, $a_i \in$ Realization Plans of CT_1, $b_i \in$ Realization Plans of CT_2, …, $n_i \in$ Realization Plans of CT_n.

Let Ua_i denote the utility of a realization plan a_i. The utility of each chosen realization plan for a component type contributes to the overall utility of a particular composition, and eventually the composite realization plan, of which the component type is a part of. If $U_{CT(ai)}$ denotes such contribution term to the utility for the composition when the realization a_i is chosen, then according to assumption 3,

$$Ua_i \geq Ua_j \Rightarrow U_{CT(ai)} \geq U_{CT(aj)};\ \forall a_i, a_j \in \mathbf{A} \ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (I)$$

The maximum utility available for the realization of a particular component type (CT_1) can be denoted as U_{CT1} and expressed as

$$U_{CT1} = \max (U_{CT(a1)}, U_{CT(a2)}, \ldots, U_{CT(ap)});\ R_{min} \leq U_{CT(ai)} \leq R_{max}\ \ \forall a_i \in \mathbf{A} \ \ldots (II)$$

In (II) R_{min} and R_{max} express the range of the values (minimum and maximum values respectively) that the utility will be evaluated to. In order to derive the utility of the composition, denoted as U_c, a function satisfying (I) can be defined as

$$U_c = f\,(U_{CT1}, U_{CT2}, \ldots, U_{CTn}, U_{prop}) \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (III)$$

where, U_{prop} is the contribution of properties (non-related to the individual components, rather related to the composition, communication among components etc.) to the utility.

In general, equation (III) can take any form, given that for each realization plan a_i, equation (I) is also maintained. A special case of equation (III) can be represented as

$$U_c = \sum_{i=1}^{n} w_i U_{CTi} + w_{n+1} U_{prop} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ (IV)$$

where, $\sum_{i=1}^{n+1} w_i = 1.0$ and each w_i indicates the relative importance (weight term) of a component type within a composition, as assigned by the developer while specifying the realization plan.

There is no restriction on the form of the utility function, as long as equation (I) is maintained. For example, utility functions calculating root mean square values or exponential functions or of any other format may also be supported. The developer has to define how he wants to establish the relation between part utilities and the utility of the composition. However, normalized values of utilities would simplify the comparison among them. Equation (IV) can be a straight-forward choice for easing the normalization, because with each part utility within the range of 0.0 and 1.0, the utility of the composition will automatically be evaluated to a value in the same range, given the sum of the weights is 1.0.

Example:

The adaptation reasoning approach can be explained with the help of the UnanticipatedTravelAssistant application from the scenarios of section 1.2. For a particular instance, the available component types, services and plans construct the variability model as presented in Figure 3.

Let us assume that at this particular instant, a significant context change[6] occurs to trigger the adaptation reasoning process. In order to illustrate the approach, the variability model of Figure 3 is enhanced by adding utility functions as presented in Figure 4. In this diagram, each atomic and service realization plan contains a utility function, derived from the QoS properties of the plan, while utility functions for composition plans are derived from part-utilities and QoS properties unrelated to a particular component (for example, communication or distribution properties). Annotations of component types are used for the sake of abbreviating texts in the equations.

Let us consider that the QoS properties of the ImageSelect plan are as follows:

```
ImageQuality = HIGH;
Memory = 100;
Images > 10000;
```

The average quality of images is determined using the ImageQualityEvaluator property evaluator[7] taking into account the Sharpness, Contrast and Distortion properties. The average may be calculated taking a sample from the total set of images. Instead of using numerical values, the quality of the image may be enumerated as HIGH, MEDIUM and LOW, based on some ranges of the numerical values. Such conversions are done by the property evaluator function. The component best suits when large number of high quality images is to be processed. For working best, it requires 100 units of memory. The developer of the component has to be aware of such properties and therefore, he has to provide the utility function and property evaluators in the ImageSelect plan.

[6] Realization plans register their context dependencies. When a particular context value change influences any of the realization plans currently in use, then the adaptation reasoning process is triggered.

[7] The detailed model of the property evaluator is presented in Figure 30.

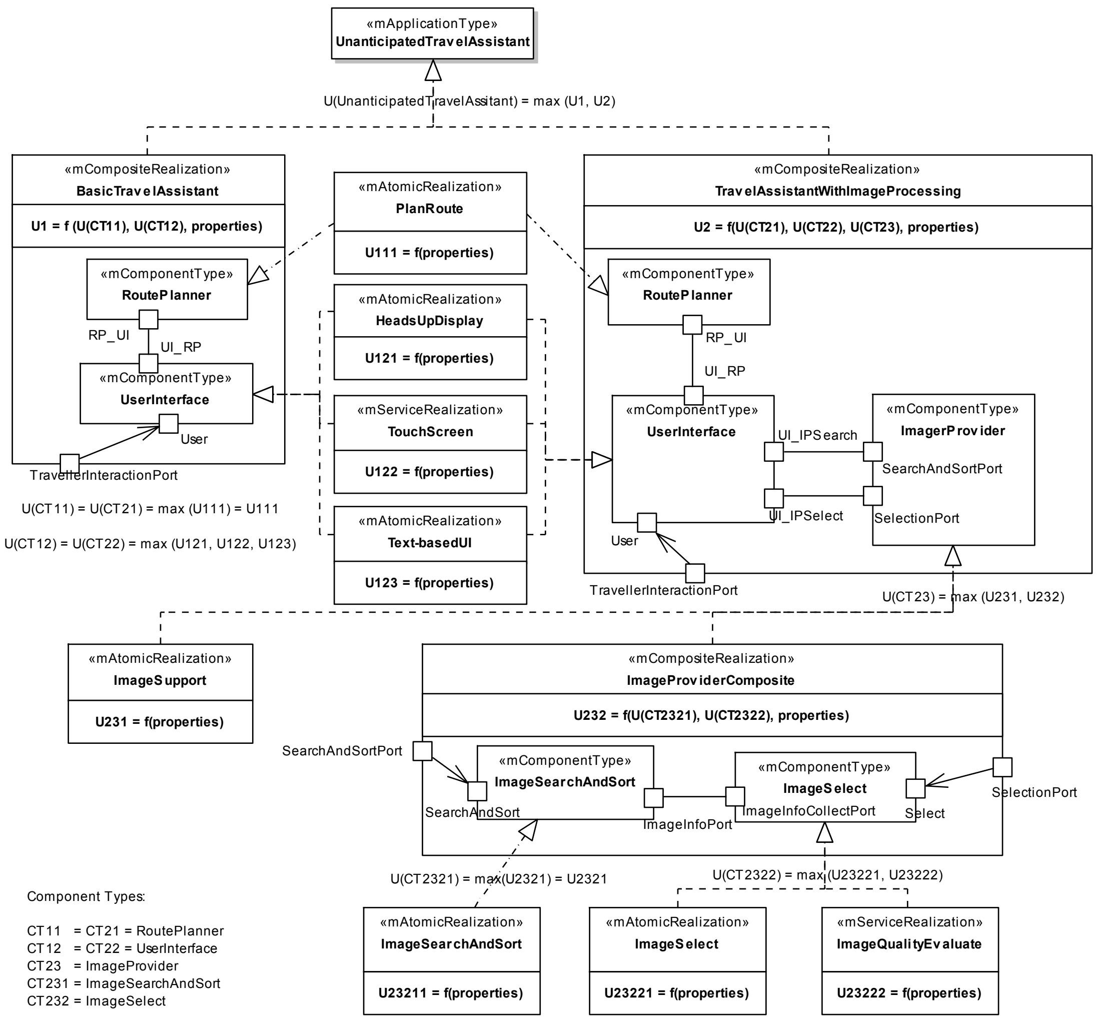

«mApplicationType» UnanticipatedTravelAssistant
U(UnanticipatedTravelAssitant) = max (U1, U2)
«mCompositeRealization» BasicTravelAssistant
U1 = f (U(CT11), U(CT12), properties)
«mComponentType» RoutePlanner
RP_UI
UI_RP
«mComponentType» UserInterface
User
TravellerInteractionPort
U(CT11) = U(CT21) = max (U111) = U111
U(CT12) = U(CT22) = max (U121, U122, U123)
«mAtomicRealization» PlanRoute
U111 = f(properties)
«mAtomicRealization» HeadsUpDisplay
U121 = f(properties)
«mServiceRealization» TouchScreen
U122 = f(properties)
«mAtomicRealization» Text-basedUI
U123 = f(properties)
«mCompositeRealization» TravelAssistantWithImageProcessing
U2 = f(U(CT21), U(CT22), U(CT23), properties)
«mComponentType» RoutePlanner
RP_UI
UI_RP
«mComponentType» UserInterface
UI_IPSearch
UI_IPSelect
User
TravellerInteractionPort
«mComponentType» ImagerProvider
SearchAndSortPort
SelectionPort
U(CT23) = max (U231, U232)
«mAtomicRealization» ImageSupport
U231 = f(properties)
«mCompositeRealization» ImageProviderComposite
U232 = f(U(CT2321), U(CT2322), properties)
SearchAndSortPort
«mComponentType» ImageSearchAndSort
SearchAndSort
ImageInfoPort
«mComponentType» ImageSelect
ImageInfoCollectPort
Select
SelectionPort
U(CT2321) = max(U2321) = U2321
U(CT2322) = max (U23221, U23222)
«mAtomicRealization» ImageSearchAndSort
U23211 = f(properties)
«mAtomicRealization» ImageSelect
U23221 = f(properties)
«mServiceRealization» ImageQualityEvaluate
U23222 = f(properties)
Component Types:
CT11 = CT21 = RoutePlanner
CT12 = CT22 = UserInterface
CT23 = ImageProvider
CT231 = ImageSearchAndSort
CT232 = ImageSelect

Figure 4: Application variability model enhanced with utility functions

Based on these QoS properties, a utility function can be defined as follows:

```
U23221 =  0.5*(
               1.0; if context.Memory > 100
               1.0 - (100 - context.Memory)/100; otherwise)
          + 0.1 * (
               1.0; if context.ImageQuality = HIGH
               0.8; if context.ImageQuality = MEDIUM
               0.3; if context.ImageQuality = LOW
               0.0; otherwise)
          + 0.4 *(
               1.0; if context.Images > 10000
               0.3; otherwise)
```

For a particular context situation (when the adaptation reasoning is done), context.Memory = 90, context.ImageQuality = HIGH and context.Images = 200 will result in a utility value of

```
U23222 = 0.5*(1.0 - (100 - 90)/100) + 0.1*(1.0) + 0.4*(0.3) = 0.67
```

The utility value is much affected by the fact that the primary target of using this component is to select from a huge number of images. However, the other factors, e.g., memory requirement and image quality requirement fit well and a utility value of 0.67 is obtained.

The utility of the service plan ImageQualityEvaluate may depend on the same set of properties or a different set, which is influenced also by the service level agreement; for example, a costly service may result in a lower utility. However, the corresponding utility, U23222 can be evaluated similarly; say, for this particular situation, the utility of this realization plan is 0.8, may be because the service is cheap and we need to select from only about 200 images.

Since U23222 is higher than U23221, the service realization ImageQualityEvaluate is favored to realize the ImageSelect component type. Now, its contribution to the composition of the ImageProviderComposite plan is, U(CT2322) = max(U23221, U23222) = 0.8.

The utility for the ImageSearchAndSort component type, U(CT2321) can be computed in the same manner. For this particular case, this will equal to U23211, because there is only a single realization possibility. However, if it is detected that a component type does not even have a single realization plan, then the complete composition plan to which this component type belongs is discarded. Let us assume that U(CT2321) = U23211 = 0.9.

Now, in the simplest case, let us assume that the utility of the ImageProviderComposite plan has a contribution of 60% from U(CT2321) and 30% from U(CT2322), while the other properties (unrelated to particular components) contribute to 10% of its utility. The property contribution can be expressed the same way using a function like the property evaluator. Let us presume that the value is 0.7. Then,

```
U232 = 0.6*U(CT2321)+0.3*U(CT2322)+0.1*0.7

     = 0.6* 0.9 + 0.3 * 0.8 +0.1*0.7

     = 0.85
```

This is a good utility; however, it may be possible that the utility of the ImageSupport realization plan, U231 is even higher, because it provides the functionalities by a single component without having to care about communication or availabilities of two different components as in the ImageProviderComposite realization plan. However, it may also be possible that this component has high resource requirement and poor computation performance etc. So, the utility depends on all these factors and can be evaluated only based on the current context situation. Let us assume that the utility is 0.8. This slightly lower value makes the ImageSupport atomic plan a worse choice. Going one step upward in the variability hierarchy, the part-utility of the ImageProvider component type in the TravelAssistantWithImageProcessing plan becomes

```
U(CT23) = max (U231, U232) = max (0.8, 0.85) = 0.85
```

Following the same procedure, the utility of the TravelAssistantWithImageProcessing plan, U2 can also be calculated. The BasicTravelAssistant realization plan also has a utility, U1 calculated from the chosen realizations for its constituent component types. If U1 > U2, BasicTravelAssistant is selected to realize the application, otherwise the

TravelAssistantWithImageProcessing plan will be used. While realizing the application, the chosen plans at different levels are considered to instantiate the components and/or to bind to the services. For example, a composition of the components and services corresponding to the PlanRoute, TouchScreen, ImageSearchAndSort and ImageQualityEvaluate plans realizes the application.

In this approach the number of times the utility function needs to be evaluated equals to the number of plans only, and not to the (possibly) huge number of all possible application variants. Also, the successful application of the approach depends on four reasonable assumptions, as introduced at the start of this section (5.3.1); but it does not apply to utility functions that violate these assumptions.

5.3.2 Meeting Resource Constraints

Each running application is allowed to use a certain amount of resources, assigned to it by an underlying middleware or operating system. Therefore, the application variant chosen by applying the reasoning approach of section 5.3.1 might not be practically realizable. This problem demands a check of resource constraints of the chosen variant against the runtime availability of the required resources. If such constraints are not met, another variant must be chosen that obviously provides lower utility; but fits within the resource constraints.

Ideally, as it is done in MUSIC, resource constraints could be checked for each of the variants before checking for their utilities; but that process would suffer from the combinatorial explosion, which we would like to avoid. Therefore, we first find a variant by applying the reasoning approach and then apply a search mechanism around the initially selected plans to find a variant that provides a feasible solution satisfying constraints for each of the resources with the minimum sacrifice to the utility.

The search is performed once for each of the resources. The target is to use a different variant for each of the individual components until the resource constraints are met. The first step in the search mechanism is to select the starting point among the chosen components for the application composition. Such a selection of the starting point of searching for checking a particular resource constraint may be done in different ways, as presented below:

- The component that requires the most amount of that resource can be a reasonable target, because a second variant of that component would most probably release an appreciable amount of resources, in a way to speed up the search.
- A second choice would be to start with the least important component so that replacing it with its second best variant would not result in much loss of utility.
- Both of the above choices have their pros and cons and a combination of them would suggest using the ratio of the resource needs to the importance of each component as the guiding factor to select the starting point.

For the starting component, an alternative is chosen, which consumes less resource than the previously chosen one, while provides the highest utility among the remaining options. For example, in Figure 4, if the TouchScreen user interface was initially chosen; but fails in fulfilling a resource constraint, then the one between the HeadsUpDisplay and the Text-basedUI user interface that provides the higher utility is chosen in this step, provided that neither of them requires more resource than the

TouchScreen user interface. If the resource saved because of selecting this new variant is still not sufficient to meet the resource constraint, then we proceed with the next component. For this case, the PlanRoute realization has no alternative; therefore, we proceed with the plans for the ImageProvider component type. After the first run, if it still requires more resources than the available limit, remaining alternatives are checked; for example, we have to go back to the remaining option for the user interface, provided that it consumes even less amount of the resource in concern. Such steps are repeated until a variant is obtained that fits within the resource limit. For example, it may even be possible that for the TravelAssistantWithImageProcessing plan, no variant was possible and the other option, i.e., the BasicTravelAssistant plan may provide a feasible solution.

The approach has the limitation that in extreme cases we might have to sacrifice utilities to a great extent and the search for resource-fitting variant may be cumbersome; but it still helps the adaptation reasoning process avoiding the combinatorial explosion. Therefore, it will provide a feasible (satisfying architectural and resource constraints) solution, if any, within a time frame of a few seconds, which is not affected by the number of application variants; rather depends on the number of plans.

5.3.3 Meeting Architectural Constraints

Like resource constraints, the MUSIC reasoning approach also checks an application variant against architectural constraints [58] before evaluating its utility. However, such checking also suffers from combinatorial explosion of the number of variants, because every variant passing the resource constraint test has to go through the architectural constraint check.

In this work, we do not alter the specification technique of architectural constraints; but the reasoning approach must be adjusted to avoid combinatorial explosion. For this purpose we adopt a similar technique as it is done for checking resource constraint. First of all, a variant is chosen applying the basic reasoning approach and afterwards misfit plans are replaced by a fitting alternative.

The initially selected variant is checked against architectural constraint and when the constraint fails at some point because of choosing a plan which is not feasible, and then among its feasible alternatives, the one with the highest utility is chosen. This process is repeated until a variant is obtained that passes all the architectural constraints.

In the application presented in the scenario of this work, we have not used architectural constraints.

5.3.4 Pros and Cons

In this approach, the number of times the utility function has to be evaluated corresponds to the number of plans and not to the number of application variants. The importance of the approach can not be revealed from simple cases like what is presented in Figure 3. In fact, for this simple case, there will be only 12 application variants with a total of 11 plans. Therefore, the straightforward approach of calculating the utility separately for each application variant, as it is done in MUSIC, would also be fine. However, such approaches suffer greatly from the combinatorial explosions and that fact has motivated us developing this new approach, as presented in this thesis. The main benefits of the approach are listed below:

- The approach is not vulnerable to scalability and therefore, it will still work where a straight forward approach will certainly fail in the case of a huge number of application variants.
- The MUSIC adaptation reasoning approach [57] [59] requires only a single utility function to be specified for the complete application. Although this reduces the modeling effort, designing a proper utility function for the complete application is quite complex. It becomes even more difficult, especially for the case of the unanticipated adaptation, where we intend to provide that level of flexibility that an individual application developer may just specify his needs of functionalities without caring much about how they will be realized or how the components realizing that application might have context dependencies. On the other hand, in this approach a particular developer may focus only on the utility function of the component he is providing.
- This approach abstracts away properties by utilities at a very early stage of the reasoning. Therefore, the need for evaluating the properties of a composite component from its constituent components is removed. This can only make the reasoning approach faster.
- Since this approach calculates the utility for each plan only once, the number of times the utility has to be evaluated is linearly dependent on the number of plans in the variability model. Thus, using the Big-Oh notation [104] the complexity of the reasoning algorithm can be expressed as O(n), where n is the number of plans. On the other hand, if a composition has c number of component types with each component type having n plans, the total number of variants from that composition will be equal to n^c. In practice the number of realization plans for different component types in a composition varies. However, the complexity of any reasoning approach that calculates the utility for individual application variants will be $O(n^c)$.

In exchange to the gained reasoning speed, the approach suffers from a few shortcomings:

- It requires providing utility function for each individual plan. The modeling effort increases, though it may be countered by the fact that many of the property evaluators as specified in the MUSIC approach are no longer needed.
- The evaluated variant may not be the perfect choice to provide the highest utility, when the complete composition of the application is concerned. The approach is governed by four assumptions (see section 5.3.1) and it might be possible that not all practical applications maintain that. For example, for a particular component type, we may have different realizations from different developers, where each of them is using a different value range for its utility function. Also, for some applications, the importance of a particular component in the composition may not always be fixed. In that case, the approach is only valid, if the weights can also be adjusted dynamically. The abstraction of properties through utilities at an early stage of adaptation reasoning may also influence obtaining a wrong result.

To summarize, the focus of the approach is the reasoning speed, especially when the size of the application variants can not be ensured to be within a certain limit. We foresee that the approach can be applied for a good range of practical applications, although the applicability is constrained by the validity of the reasonable assumptions (section 5.3.1) adopted for its development.

6 Middleware

The middleware supports the unanticipated adaptation to applications through providing a number of middleware services. In a ubiquitous computing environment, there can be a number of devices running any arbitrarily large number of applications on a large number of middleware instances. Therefore, here we delineate the scope of an adaptation, by defining an adaptation domain.

As also in MUSIC, an adaptation domain is defined a collection of U-MUSIC middleware instances controlled by one adaptation manager[8]. It includes one MASTER node (normally a handheld device) which is bound to a user and acts as the nucleus around which the adaptation domain forms dynamically as SLAVE nodes come and go. The dynamic change of an adaptation domain is caused by the movement of the MASTER node or changes in connectivity due to other phenomena. Figure 5 presents an example of an adaptation domain corresponding to the scene 2 of section 1.2.2. At a particular instant the UnanticipatedTravelAssistant application is using the Map Downloader component from Stephan's device and the TouchScreenUI service offered by the coffee machine at the petrol station. The adaptation domain consists of the usable components and services for the particular application. Note that the MP3 component of Stephan's device is not included in the adaptation domain, because it is not of interest for the UnanticipatedTravelAssistant application. On the other hand, the TextToSpeech component resides in the adaptation domain, because it is usable for the application, although it is not used for the particular composition.

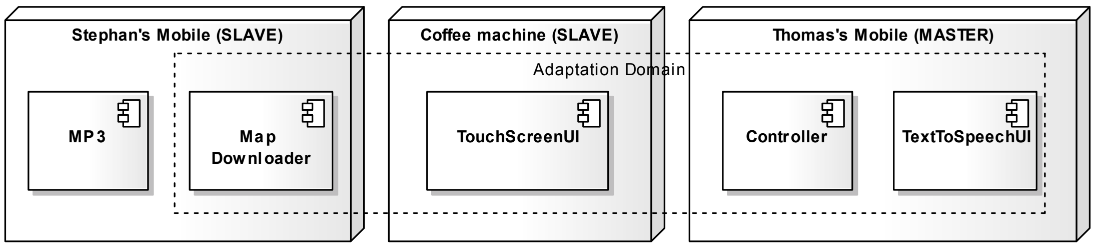


Figure 5: Adaptation domain

Adaptation domains may overlap in the sense that a SLAVE node may be a member of more than one adaptation domain. This adds to the dynamics and increases the complexity because the amount of resources the auxiliary nodes are willing to provide to a particular domain may vary depending on the needs of other domains which they are also serving.

The notion of component type and service differs slightly in this work from MUSIC. In MUSIC, a realization plan is bound to a particular component type, which makes it impossible to use components from other developers, if they do not specify the same type name. We consider that any node on the adaptation domain that is running an

[8] Adaptation manager is a component of the U-MUSIC middleware.

instance of the U-MUSIC middleware may provide components with associated plan descriptions, as defined by the U-MUSIC conceptual meta-model. Such plans are matched with the types at runtime based on the functionalities. Like the service ontology in MUSIC, we define a functionality ontology, where the top level ontology can be extended by individual developers to create sub-ontologies for their types and plans. In U-MUSIC, services are provided by non-U-MUSIC applications/nodes using protocols for service discovery, binding and communication. In the ongoing development, MUSIC nodes are also considered to host services to be used by both MUSIC and non-MUSIC applications. In this work, we do not consider service hosting by U-MUSIC applications.

The user of a MASTER node may start (instantiate) and stop (remove) U-MUSIC applications, and the set of running applications inside the adaptation domain is adapted by the adaptation manager in accordance with these user actions, relevant context changes, and resource constraints.

Adaptation involves binding the component types of the application by instantiating appropriate component implementations inside the adaptation domain, where a system is built, or outside (external service) the adaptation domain. In the first case, the adaptation manager has control of the resources. In the latter case, this is outside the control of the adaptation manager, and it is necessary to negotiate a service level agreement (SLA) with the provider to be able to reason about the suitability of different providers. External services may be provided by non-U-MUSIC systems.

In the following subsections, we present the middleware architecture along with some implementation issues. The work is based on the MUSIC middleware [2] and therefore, we will first present a complete overview at the very top level of the middleware. However, for this work, only a few middleware components are updated and only those components will be discussed in more details.

6.1 Middleware Architecture

The layered view of the U-MUSIC runtime environment is presented in Figure 6. The main intentions for organizing the architecture into layers are portability and separation of concerns. With respect to the portability issue, the core services and the system-level services of the U-MUSIC architecture encapsulate the heterogeneity of the underlying technologies (e.g. the OSGi framework [75]) as well as the varying computing and communication infrastructure. By using interfaces provided by these services, components in higher layers are not affected when, for example, the networking technologies change. This increases the portability and reusability of the components in the higher layers. With respect to the separation of concerns issue, the system-level services provide services which crosscut the modular structure (in the higher layers) of the system.

The relation among layers in a layered architecture is ‘allowed to use’. In general, the usage in layers of the middleware architecture flows downward, as defined below:

- Components in the same layer are allowed to use each other. For example, the Adaptation Middleware can use services provided by the Context Middleware.

- A layer is allowed to use not only the layer below, but also any lower layer. For example, the Application can access information about resource availability offered by the Resource Manager.

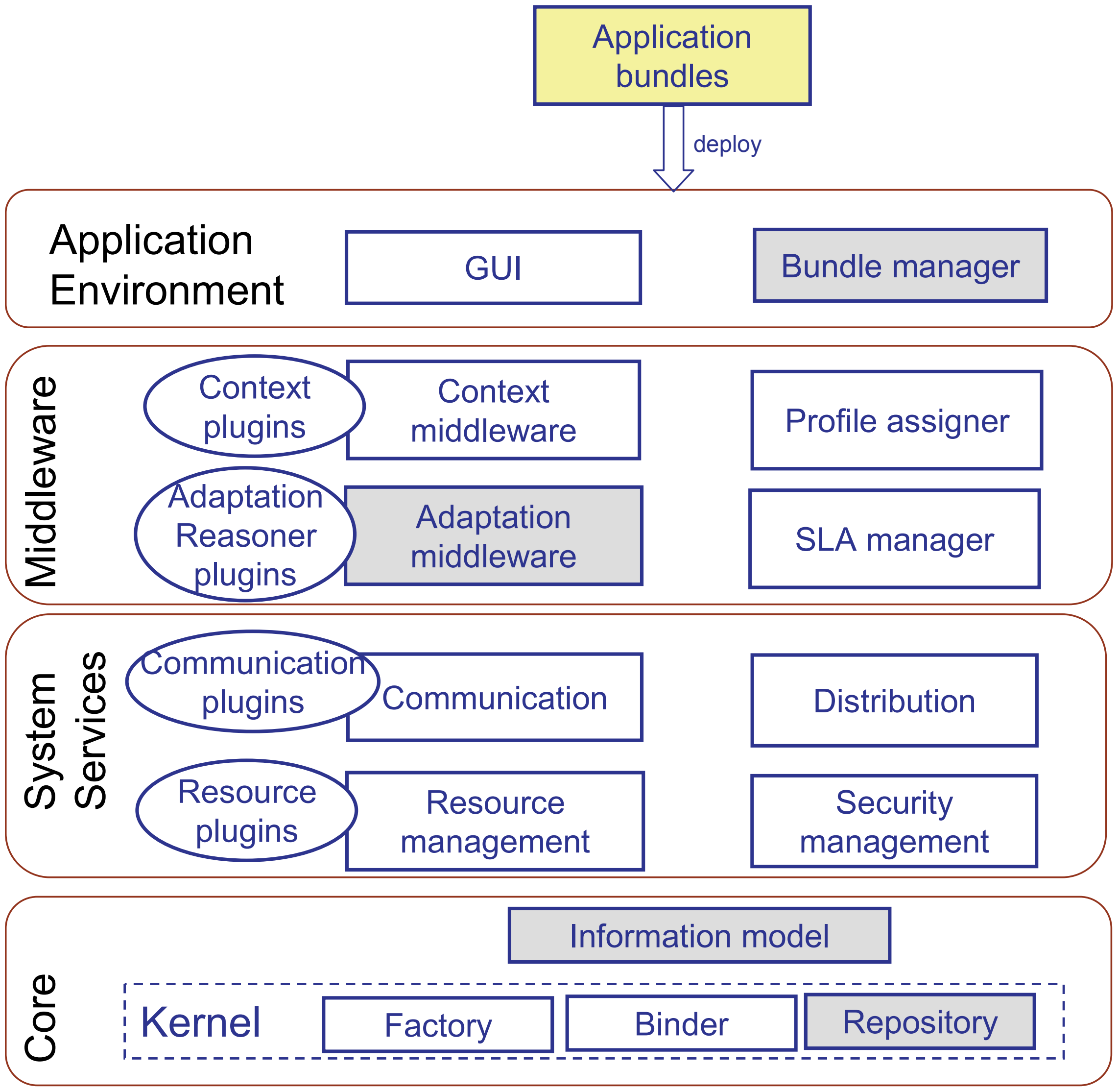


Figure 6: Layered view of the U-MUSIC runtime environment

The core consisting of the minimum set of components required to instantiate the middleware represents the backbone of the U-MUSIC platform. It contains a set of services, also referred to as the Kernel, which jointly provide the low-level operations to deploy and to easily retrieve the various kind of services (either middleware services or application components) hosted by the U-MUSIC platform. These core services include the components Factory, Repository, and Binder. The component Factory provides an interface for managing the life cycle of a service. The Factory component implements the so-called design pattern described by [86]. The Binder component basically provides mechanisms to connect two references of services. The binding can be either local (setting a local reference) or remote (deploying a connector) depending on the behavior implemented by the component Binder. The binding component applies the paradigm promoted by Binding frameworks. The component Repository stores the list of services available for the technology supported by the capsule and provides facilities for adding, removing and listing these services. The component Repository provides also an interface (IResolver) to retrieve the reference of a given service. Since services are indexed in the Kernel using static properties, these static properties are also used to retrieve the available references. The component Repository can be considered as a lightweight implementation of the Trading service principles. The Information Model contains the data structures for the U-MUSIC data types.

The system services block groups together system-level services which encapsulate the heterogeneity of the underlying computing and communication infrastructure. The communication service provides support for searching and binding of remote components and services, as well as for exporting local components and services. The distribution service is designed for the exchange of arbitrary information types between networked hosts. The U-MUSIC middleware leverages distribution service for distributing context information among different U-MUSIC nodes. The Resource Management service is the U-MUSIC component responsible for managing in a centralized way the low-level resources available in an adaptation domain. The Security Management service provides middleware-level security management for the middleware services.

The Middleware environment block collects a set of services providing the core capabilities of the U-MUSIC middleware in terms of context awareness, self-adaptation and SLA negotiation capabilities. The Context Middleware component is responsible for collecting, organizing, managing and sharing the context information, with the ultimate goal of making it available to context clients. The Context Middleware is primarily used by the Adaptation Middleware, but additionally it can be used directly by context-aware applications. The Adaptation Middleware is responsible for reasoning on the impact of context changes on the application(s), and for adapting the set of running applications so that they best fit the current context and resource situation. The Profile Assigner allows the support of a dynamic platform configuration to optimize the scarce resources of mobile devices. The SLA Manager enriches the U-MUSIC middleware with negotiation capabilities, to enable the incorporation of services with associated QoS levels into the adaptation mechanisms.

The Application block of the U-MUSIC platform groups two services, responsible for the management of the U-MUSIC bundles (Bundle Manager) and for providing a Graphical User Interface for the management of the U-MUSIC middleware (GUI component) and deploying U-MUSIC bundles.

The architecture is pluggable; i.e., new plug-ins can be added and removed as per requirement. For example, new context sensors and reasoners can be plugged in (instantiated) corresponding to particular context information. Also, there can be a number of adaptation reasoners implementing different adaptation reasoning approaches. Similar situations may occur also for resource and communication plug-ins.

For this work, we have adopted the MUSIC middleware as the baseline and updated the components Bundle Manager, Adaptation Middleware, Information Model and Repository, as they are indicated by filled boxes in Figure 6. In the following we describe these components emphasizing the updates, while for a detailed description of the MUSIC middleware architecture, we refer to MUSIC deliverables D4.2 [60] and D4.3[9].

6.1.1 Information Model

The Information Model contains the data structures for different MUSIC data types. These data types, e.g., Plan, IBundle, ComponentType etc. are used to represent the MUSIC variability model which is the basis for taking adaptation decision and reconfiguration of the application. It also provides the interfaces to retrieve such information from deployed bundles.

Interfaces

Figure 7 gives the overview of the provided interfaces for the Information Model while the details of these interfaces are given in Figure 8.

The IBundle interface defines the methods utilized to publish the information elements contained in a MUSIC bundle, including application types, component types, plans and extension plans.

IPlan defines the interface shared by all plans. This interface is used to retrieve information like functionalities realized by the component corresponding to the plan, dependencies to context information along with plan name, factory name[10], number of variants created for a particular plan etc.

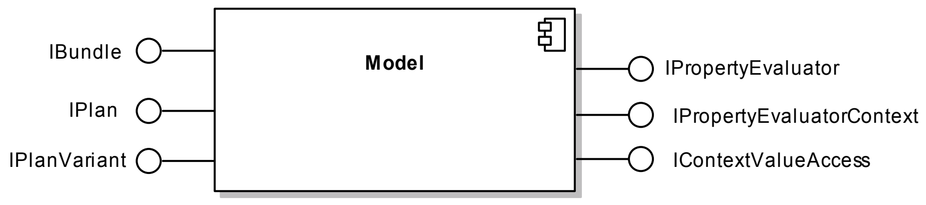


Figure 7 Interface overview of the Information Model

IPlanVariant defines the interface for accessing a variant[11] for a plan. Using this interface the set of properties, parameter settings, resource needs etc. can be retrieved. Also, a reference to the plan that this variant is a part of can be retrieved. A plan can have a set of plan variants. The plan defines the common information while the plan variants specify some additional information related to the plan. A plan variant consists of a set of properties, features, parameter settings, resource needs and device settings for this variant. For a service plan, the plan variants specify different service instances.

---

[9] It will also be available soon on the MUSIC website: http://www.ist-music.eu/MUSIC/results/music-deliverables

[10] The name returned from this method can be resolved to an IFactory. The value of this name can be e.g. “OSGi”, “JavaRMI”, “WebService” etc, and refers to different technologies supported.

[11] A PlanVariant indicates a plan with a particular set of properties, parameter setting, resource requirements etc. Though it is not introduced in the conceptual meta-model, we use the concept in the middleware.

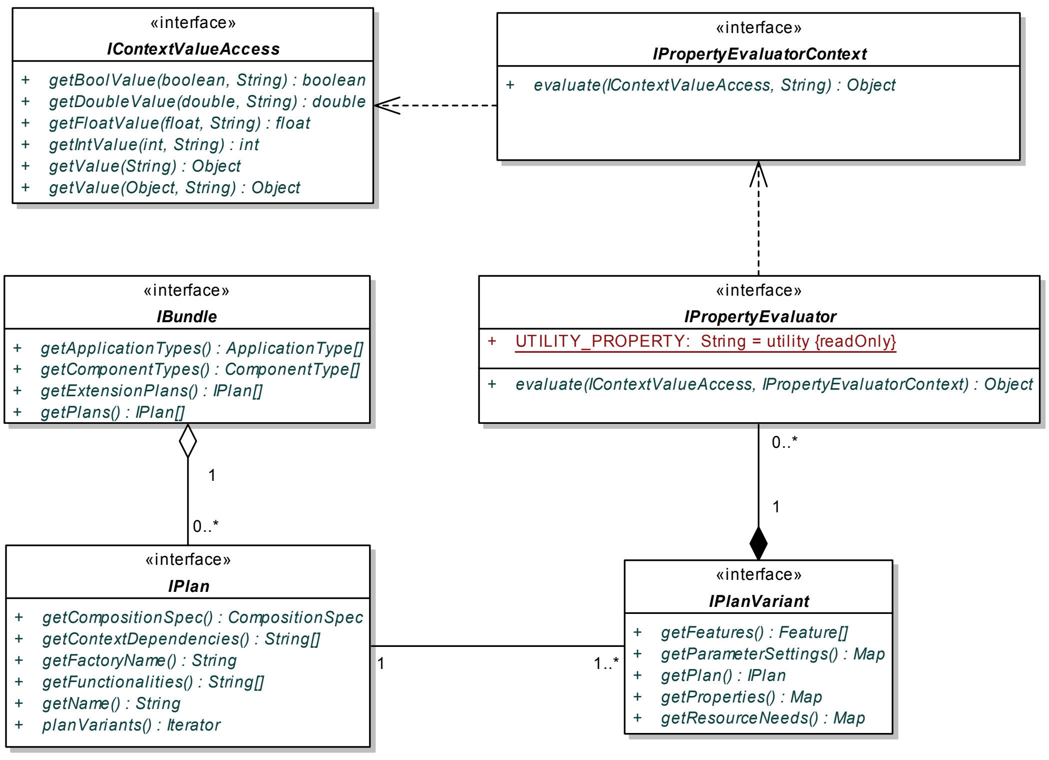

«interface»
IContextValueAccess
+ getBoolValue(boolean, String) : boolean
+ getDoubleValue(double, String) : double
+ getFloatValue(float, String) : float
+ getIntValue(int, String) : int
+ getValue(String) : Object
+ getValue(Object, String) : Object
«interface»
IPropertyEvaluatorContext
+ evaluate(IContextValueAccess, String) : Object
«interface»
IBundle
+ getApplicationTypes() : ApplicationType[]
+ getComponentTypes() : ComponentType[]
+ getExtensionPlans() : IPlan[]
+ getPlans() : IPlan[]
«interface»
IPropertyEvaluator
+ UTILITY_PROPERTY: String = utility {readOnly}
+ evaluate(IContextValueAccess, IPropertyEvaluatorContext) : Object
1
0..*
0..*
1
«interface»
IPlan
+ getCompositionSpec() : CompositionSpec
+ getContextDependencies() : String[]
+ getFactoryName() : String
+ getFunctionalities() : String[]
+ getName() : String
+ planVariants() : Iterator
1
1..*
«interface»
IPlanVariant
+ getFeatures() : Feature[]
+ getParameterSettings() : Map
+ getPlan() : IPlan
+ getProperties() : Map
+ getResourceNeeds() : Map

Figure 8 Interface description of the Information Model

The IPropertyEvaluator interface is mainly used to provide a method that evaluates the value of a property, based on the current context values, and in the specified evaluator context. This method is typically called to evaluate a property associated with a plan in its current context of use. The complexity of such methods can vary, and the simplest method can e.g. just return a constant independent of any context. More complex methods can calculate the value depending on values that are currently stored in the context repository (e.g. originating from sensors) and on values which are found by calling other evaluator methods through the property evaluator context. A special case of property evaluators is the utility function. In U-MUSIC, supporting the adaptation reasoning approach of section 5.3.1, every plan variant contains at least the utility function along with any number of other property evaluator.

The IPropertyEvaluatorContext interface is called by IPropertyEvaluator to evaluate the property. It provides the evaluate() method, which evaluates a property belonging directly to this evaluator context. This method is typically called from property evaluators in order to evaluate another property belonging to the same plan.

The IContextValueAccess interface gives access to context values from the property evaluator functions. The interface only gives access to those context values to which the plans containing the property evaluators have declared dependencies.

Classes implementing the above interfaces are closely related to the conceptual meta-model presented in Figure 1.

Extensions to MUSIC

Compared to the MUSIC middleware (version 0.2.2), the following updates are made:

- The IPlan interface is updated by removing the getComponentType() method, because a plan is no longer statically related to a particular component type. It is also enhanced by adding the getFunctionalities() method in order to retrieve corresponding meta-information that can be used to match a plan to component types and/or application types at runtime.
- The evaluateForRole() method is removed from the IPropertyEvaluatorContext interface. This corresponds to the new adaptation reasoning approach (section 5.3), which does not require evaluating properties of a composition through corresponding properties of its constituent component types.
- Corresponding to the update in the conceptual meta-model (section 4.1), Role is no longer used in the information model.
- IPlanVariant contains at least one IPropertyEvaluator (utility function) and therefore, the multiplicity of 0..* is updated to 1..*.
- Classes implementing the interfaces are updated in relation to the conceptual meta-model of Figure 1.

6.1.2 Bundle Manager

The Bundle Manager is responsible for managing the U-MUSIC bundles[12] and update/install the U-MUSIC applications. This management task could be started by a user or automatically by other applications.

A U-MUSIC bundle is a flexible deployment unit which allows us to deploy individual components as well as full applications, and which also allows us to download meta-information (plans) separately from source code.

Usually, a U-MUSIC application will be bundled to make easier its deployment on the middleware. This deployment unit will be a JAR file that will contain the model of the application, the plans or descriptions that can realize each component and service of the application, as well as the Java classes and other types of resources.

However, a U-MUSIC bundle does not have to provide all the artifacts required to run a full application. It may provide just a subset and the remaining artifacts may be deployed by other bundles.

Interfaces

The diagram of Figure 9 lists the interfaces provided and required by the Bundle Manager component.

[12] Please note that although the term 'Bundle' is akin to the OSGi 'Bundle' and we use the OSGi framework for bundle management, a U-MUSIC bundle does not implement the OSGi Bundle interface; rather it implements the U-MUSIC IBundle interface and therefore, it is not analogous to an OSGi Bundle.

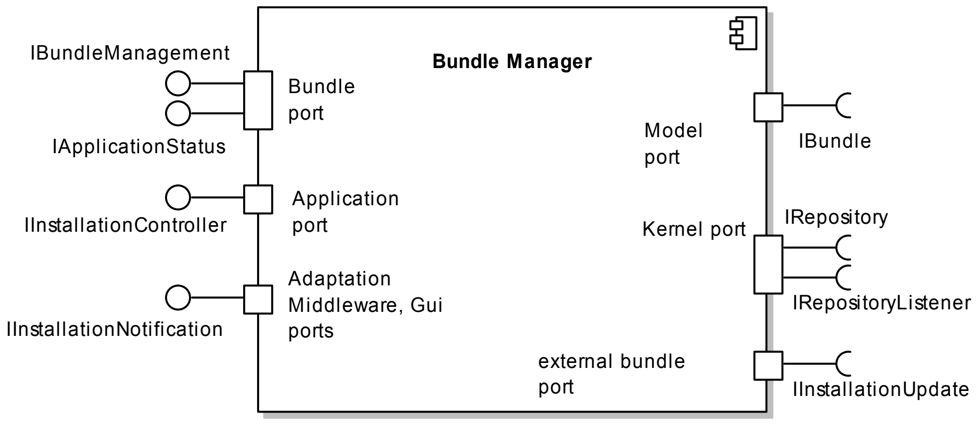


Figure 9: Interface overview of the Bundle Manager component

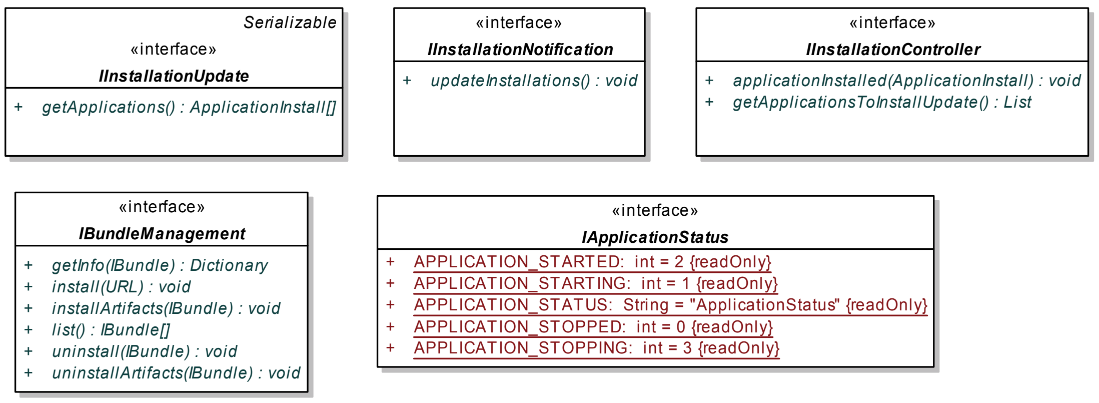

Serializable
«interface»
IInstallationUpdate
+ getApplications() : ApplicationInstall[]
«interface»
IInstallationNotification
+ updateInstallations() : void
«interface»
IInstallationController
+ applicationInstalled(ApplicationInstall) : void
+ getApplicationsToInstallUpdate() : List
«interface»
IBundleManagement
+ getInfo(IBundle) : Dictionary
+ install(URL) : void
+ installArtifacts(IBundle) : void
+ list() : IBundle[]
+ uninstall(IBundle) : void
+ uninstallArtifacts(IBundle) : void
«interface»
IApplicationStatus
+ APPLICATION_STARTED: int = 2 {readOnly}
+ APPLICATION_STARTING: int = 1 {readOnly}
+ APPLICATION_STATUS: String = "ApplicationStatus" {readOnly}
+ APPLICATION_STOPPED: int = 0 {readOnly}
+ APPLICATION_STOPPING: int = 3 {readOnly}

Figure 10: Interface description of the Bundle Manager component

The management of the MUSIC bundles is realized with the following interfaces:

- IBundleManagement is the interface which allows the management of the U-MUSIC bundles which provide the MUSIC artifacts to the runtime middleware. Figure 10 shows the methods offered by this interface. It is possible to install a bundle available in a remote node (i.e. in a web server), to make its artifacts available in the local node. The installArtifacts() method will install the artifacts provided by a specific U-MUSIC bundle. The list() method will return a list of the MUSIC bundles registered in the platform.

- IApplicationStatus is used to define constants for the different status of a U-MUSIC application.

- IRepository is a required interface, which is provided by the Kernel, to access the different repositories: application types, component types and plans repositories. All the artifacts contained in a bundle are registered in the appropriate kernel repository to make them available to the rest of the middleware.

- IRepositoryListener is a required interface, provided by Kernel, which provides the low level mechanisms for being notified of changes in a repository.

- IBundle is another required interface, which is provided by the Information Model component in order to get access to the bundle artifacts: application types, component types and plans.

The installation and update of applications is managed by the following interfaces:

- IInstallationUpdate is the interface provided by the Installer service which simplifies the management of applications (the installation of new applications and their update). The Installer service provides the list of available applications, and mechanisms to install and update them in the platform.
- IInstallationController is an interface provided by the Installation Manager to manage the MUSIC installations/updates.
- IInstallationNotification is an interface which notifies about new installations and updates available, as well as those ones that are not available anymore.

Structure

Figure 11 depicts the structure of the Bundle Manager which is composed of two different components: the Bundle Manager and the Installation Manager.

The Bundle Manager component performs the installation and un-installation of the U-MUSIC bundles. It detects all the U-MUSIC bundles through the required interface IBundle which exposes the U-MUSIC artifacts of the bundle. These artifacts are registered or unregistered in the appropriate repository hosted by the kernel:

- The application type repository contains the list of U-MUSIC applications.
- The component type repository stores all the component types.
- The plan repository includes all the plans (atomic, service and composition plans) of the platform.

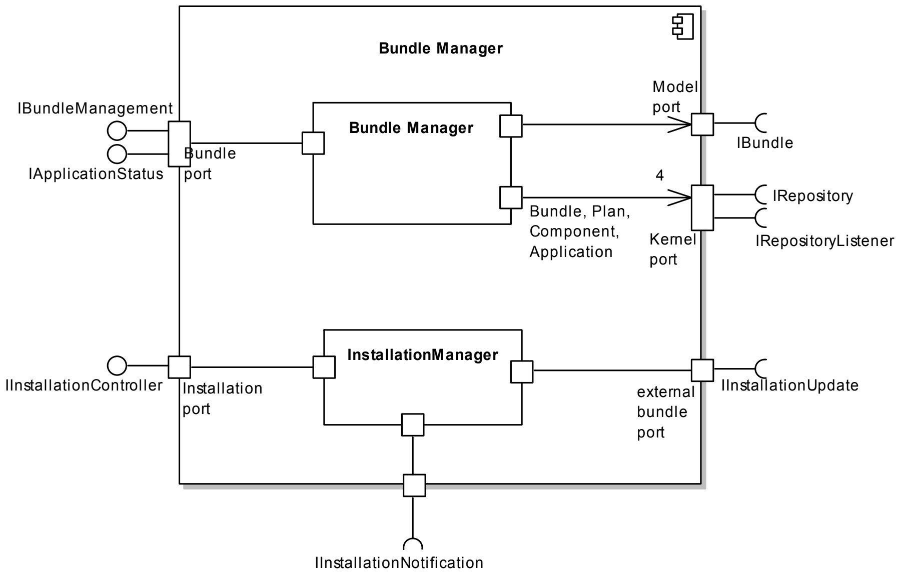


Figure 11: Structure of the Bundle Manager component

The Installation Manager interacts with the Installer service (typically a remote service) in order to simplify the installation and update of applications. The Installation Manager may notify other middleware components about the appearance of new applications (ready for the installation in the platform) and about new updates.

Behavior – installation of bundles

The bundles installation procedure is updated from that in MUSIC in order to support matching of types and plans at runtime. Figure 12 represents the sequence diagram for the installation of a U-MUSIC bundle. The bundle is available at a URL and an external agent requests the Manager to install the bundle in this location. The Bundle Manager will iterate through the artifacts contained in the bundle (through the IBundle interface) in order to register them in the kernel repositories.

For installing each of the application types contained in the bundle, first of all, all plans in the plan repository are checked, if there is any plan matching the application type in concern. Each of the matching plans is registered against the type name of the application type. Afterwards, the application type itself is registered in the application type repository after adding the application status (STARTED, STOPPED, SUSPENDED etc.) as a property.

Similar steps are followed for each of the component types in the bundle. However, they do not need any property update. In order to register plans contained in the bundle, first of all, both the application type repository and the component type repository are searched for matching types. For each of the matching types, an entry is registered in the plan repository, where the plan is registered against the type name. If it happens that no existing type matches with the plan, then it is still registered against a default String ("NOTMATCHED") indicating that no matching type for the plan is still found. However, when a new bundle is discovered that contains a matching type, this plan can be then registered against that type.

The un-installation of a bundle involves unregistering all the bundle artifacts and updating all the three repositories.

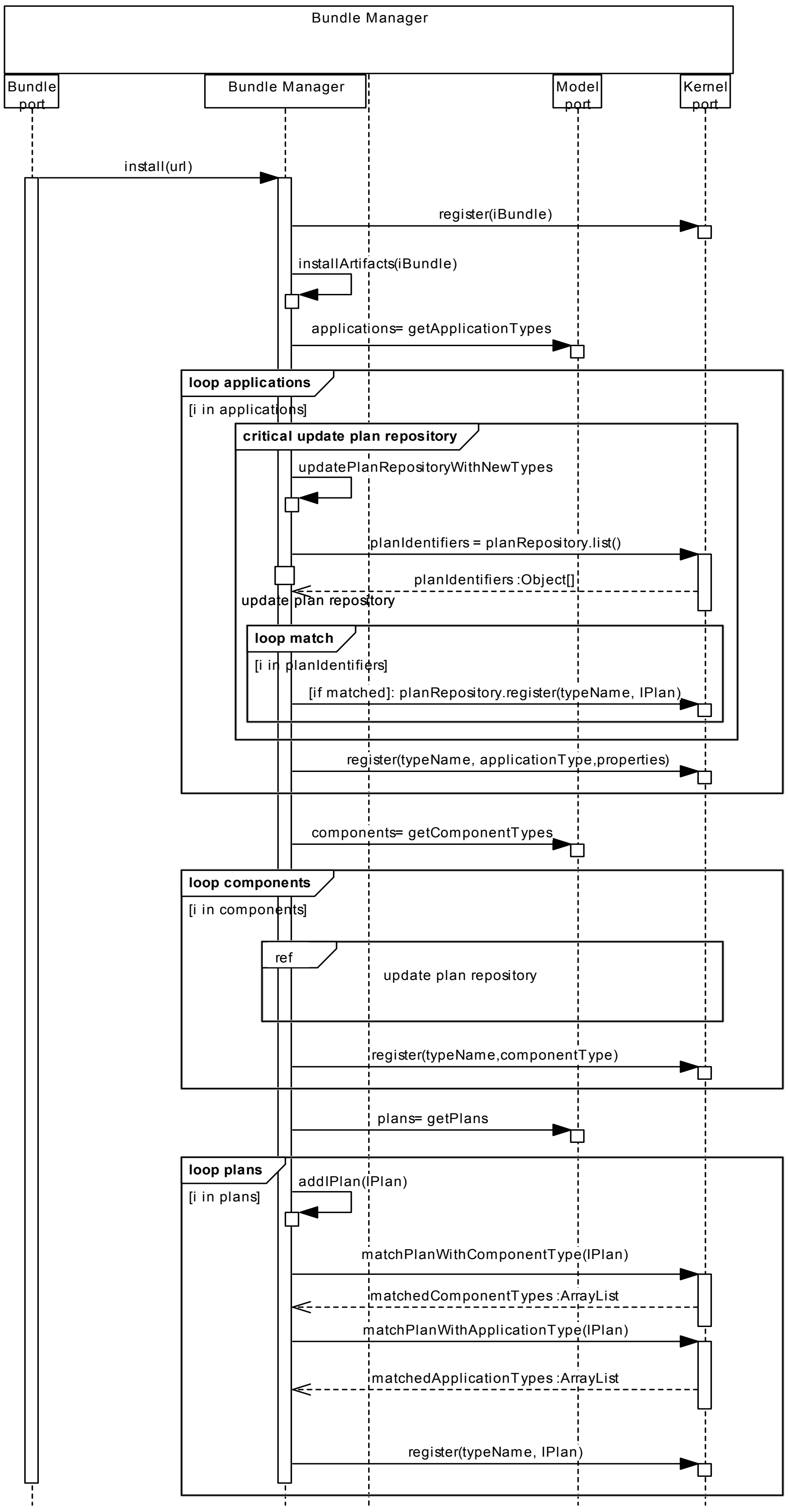

Bundle Manager
Bundle port
Bundle Manager
Model port
Kernel port
install(url)
register(iBundle)
installArtifacts(iBundle)
applications= getApplicationTypes
loop applications
[i in applications]
critical update plan repository
updatePlanRepositoryWithNewTypes
planIdentifiers = planRepository.list()
planIdentifiers :Object[]
update plan repository
loop match
[i in planIdentifiers]
[if matched]: planRepository.register(typeName, IPlan)
register(typeName, applicationType,properties)
components= getComponentTypes
loop components
[i in components]
ref
update plan repository
register(typeName,componentType)
plans= getPlans
loop plans
[i in plans]
addIPlan(IPlan)
matchPlanWithComponentType(IPlan)
matchedComponentTypes :ArrayList
matchPlanWithApplicationType(IPlan)
matchedApplicationTypes :ArrayList
register(typeName, IPlan)

Figure 12: Sequence diagram for the registration of a MUSIC bundle

Extensions to MUSIC

Compared to MUSIC, none of the interfaces is updated. However, the bundle registration process is improved supporting a runtime matching technique between types and plans. Therefore, a few new methods are added in the BundleManager Class, as depicted in Figure 13. Also, the Bundle registration process is updated, which is already explained in Figure 12.

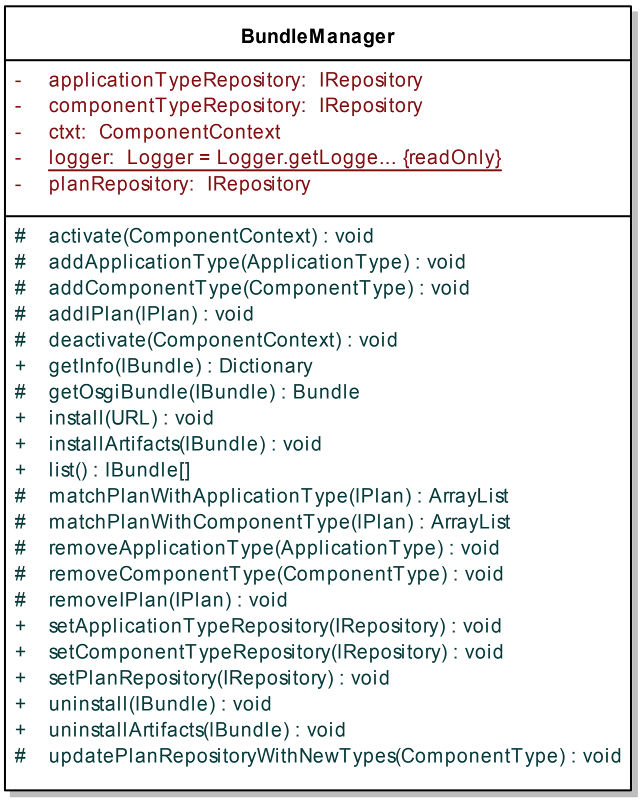

BundleManager
- applicationTypeRepository: IRepository
- componentTypeRepository: IRepository
- ctxt: ComponentContext
- logger: Logger = Logger.getLogge... {readOnly}
- planRepository: IRepository
# activate(ComponentContext) : void
# addApplicationType(ApplicationType) : void
# addComponentType(ComponentType) : void
# addIPlan(IPlan) : void
# deactivate(ComponentContext) : void
+ getInfo(IBundle) : Dictionary
# getOsgiBundle(IBundle) : Bundle
+ install(URL) : void
+ installArtifacts(IBundle) : void
+ list() : IBundle[]
# matchPlanWithApplicationType(IPlan) : ArrayList
# matchPlanWithComponentType(IPlan) : ArrayList
# removeApplicationType(ApplicationType) : void
# removeComponentType(ComponentType) : void
# removeIPlan(IPlan) : void
+ setApplicationTypeRepository(IRepository) : void
+ setComponentTypeRepository(IRepository) : void
+ setPlanRepository(IRepository) : void
+ uninstall(IBundle) : void
+ uninstallArtifacts(IBundle) : void
# updatePlanRepositoryWithNewTypes(ComponentType) : void

Figure 13: Contents of the BundleManager Class

Among the methods of the BundleManager class matchPlanWithApplicationType(), matchPlanWithComponentType(), updatePlanRepositoryWithNewTypes() are added in this work, while some other methods like addIPlan(), addApplicationType(), addComponentType() etc. are updated.

6.1.3 Adaptation Middleware

The Adaptation Middleware is responsible for reasoning on the impact of context changes on the application(s), and for adapting the set of running applications. To accomplish this, it will first select the application variant that best fit the current context, and then perform a controlled reconfiguration of the application components.

Interfaces

Figure 14 shows an overview of the interfaces provided and required by the Adaptation Middleware. The Adaptation Middleware depends on many of the services defined by other parts of the middleware. Among the interfaces shown in the figure, ITemplateBuilder and IAdaptationReasonerService and IAdaptationController are defined by the Adaptation Middleware.

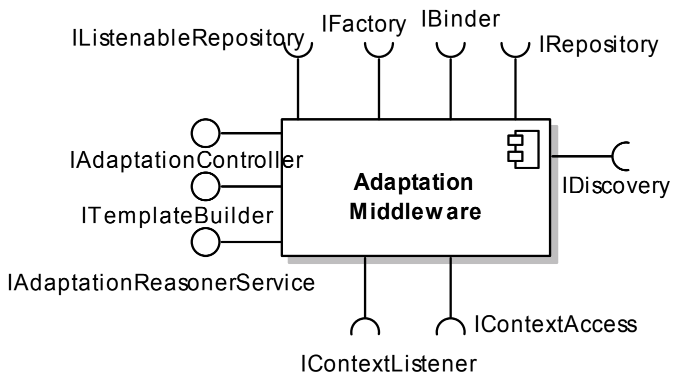
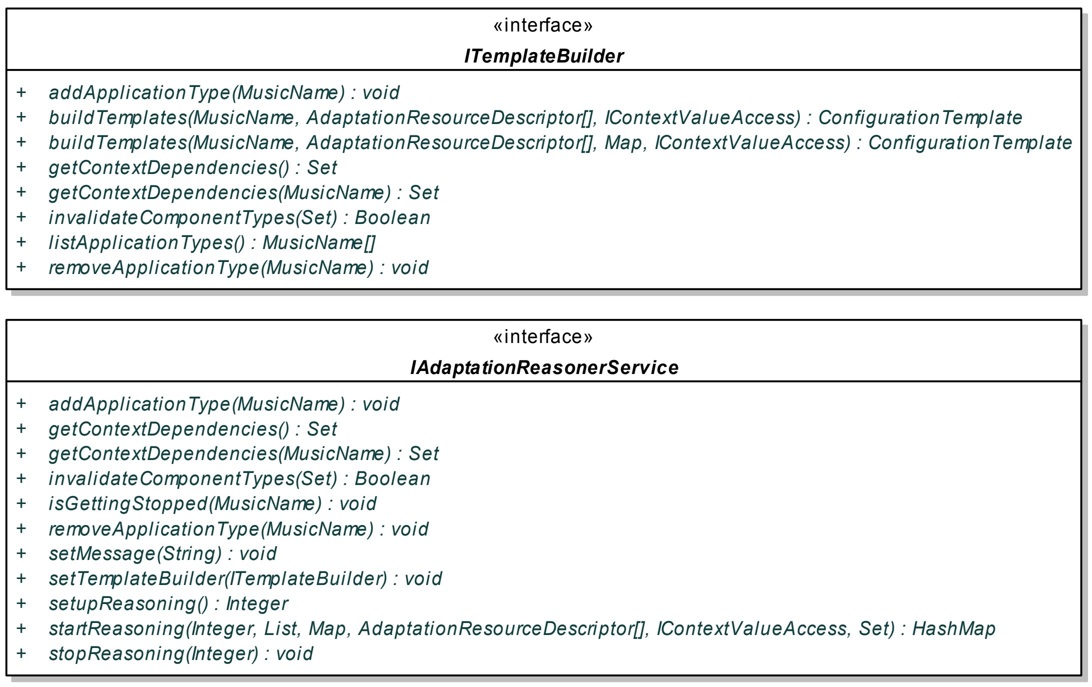


Figure 14 Interface overview of the Adaptation Middleware

The IAdaptationController interface serves as a marker interface for the Adaptation Controller and is used to determine whether the Adaptation Controller is available.

The ITemplateBuilder interface is provided by the Adaptation Middleware and can be used by the Adaptation Reasoning Services to iterate over the plans in the plan repository. Unlike MUSIC, the TemplateBuilder Class implementing the ITemplateBuilder interface returns only a single template; i.e., some of tasks of the adaptation reasoning process are delegated to the TemplateBuilder.

The IAdaptationReasonerService interface is provided by the Adaptation Middleware and it aids the TemplateBuilder in the adaptation reasoning process.



```
«interface»
ITemplateBuilder
+ addApplicationType(MusicName) : void
+ buildTemplates(MusicName, AdaptationResourceDescriptor[], IContextValueAccess) : ConfigurationTemplate
+ buildTemplates(MusicName, AdaptationResourceDescriptor[], Map, IContextValueAccess) : ConfigurationTemplate
+ getContextDependencies() : Set
+ getContextDependencies(MusicName) : Set
+ invalidateComponentTypes(Set) : Boolean
+ listApplicationTypes() : MusicName[]
+ removeApplicationType(MusicName) : void

«interface»
IAdaptationReasonerService
+ addApplicationType(MusicName) : void
+ getContextDependencies() : Set
+ getContextDependencies(MusicName) : Set
+ invalidateComponentTypes(Set) : Boolean
+ isGettingStopped(MusicName) : void
+ removeApplicationType(MusicName) : void
+ setMessage(String) : void
+ setTemplateBuilder(ITemplateBuilder) : void
+ setupReasoning() : Integer
+ startReasoning(Integer, List, Map, AdaptationResourceDescriptor[], IContextValueAccess, Set) : HashMap
+ stopReasoning(Integer) : void
```

Figure 15 Interface description of the Adaptation Middleware

The interfaces IFactory, IBinder, IRepository and IListenableRepository are provided by the Kernel component. The IContextListener interface is provided by the Context

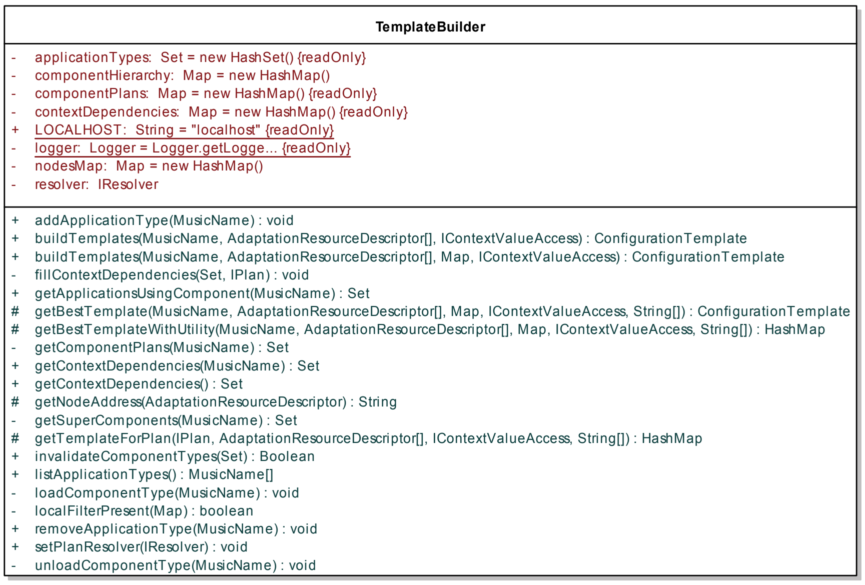
Middleware in order to receive asynchronous call-backs from the Context Middleware. The AdaptationController class, which provides the starting point of adapting an application, implements the IContextListener interface. The IContextAccess interface is also provided by the Context Middleware to access context services. The IDiscovery interface is provided by the Communication component for the publication and discovery of devices and services.

Figure 15 shows the methods defined by the ITemplateBuilder and the IAdaptationReasoner interfaces.

The details of the methods in those interfaces are not described in this work (see D4.2 and D4.3 of MUSIC [60]); however in the following we focus on the discussion of Classes that are most important for supporting the new adaptation reasoning approach (see section 5.3).

Template builder

A Template Builder allows for the iteration of all possible realizations of all available applications. It produces configuration templates by iterating over the variation space of an application. It uses the plan repository to find all plans for an application and can contain application specific heuristics to limit the number of variants. It is deployed on each node. For the adaptation reasoning approach, the TemplateBuilder Class is updated considerably and therefore, more details of the Class are shown in Figure 16.



```
TemplateBuilder

-  applicationTypes:  Set = new HashSet() {readOnly}
-  componentHierarchy:  Map = new HashMap()
-  componentPlans:  Map = new HashMap() {readOnly}
-  contextDependencies:  Map = new HashMap() {readOnly}
+  LOCALHOST:  String = "localhost" {readOnly}
-  logger:  Logger = Logger.getLogge... {readOnly}
-  nodesMap:  Map = new HashMap()
-  resolver:  IResolver

+  addApplicationType(MusicName) : void
+  buildTemplates(MusicName, AdaptationResourceDescriptor[], IContextValueAccess) : ConfigurationTemplate
+  buildTemplates(MusicName, AdaptationResourceDescriptor[], Map, IContextValueAccess) : ConfigurationTemplate
-  fillContextDependencies(Set, IPlan) : void
+  getApplicationsUsingComponent(MusicName) : Set
#  getBestTemplate(MusicName, AdaptationResourceDescriptor[], Map, IContextValueAccess, String[]) : ConfigurationTemplate
#  getBestTemplateWithUtility(MusicName, AdaptationResourceDescriptor[], Map, IContextValueAccess, String[]) : HashMap
-  getComponentPlans(MusicName) : Set
+  getContextDependencies(MusicName) : Set
+  getContextDependencies() : Set
#  getNodeAddress(AdaptationResourceDescriptor) : String
-  getSuperComponents(MusicName) : Set
#  getTemplateForPlan(IPlan, AdaptationResourceDescriptor[], IContextValueAccess, String[]) : HashMap
+  invalidateComponentTypes(Set) : Boolean
+  listApplicationTypes() : MusicName[]
-  loadComponentType(MusicName) : void
-  localFilterPresent(Map) : boolean
+  removeApplicationType(MusicName) : void
+  setPlanResolver(IResolver) : void
-  unloadComponentType(MusicName) : void
```

Figure 16: The TemplateBuilder Class

While iterating over the plans, it performs the adaptation reasoning task. Unlike MUSIC, where the TemplateBuilder provides all the possible templates corresponding to different application variants to the AdaptationReasonerService, in our case it will return only a single template, which will be passed to the AdaptationController through the AdaptationReasonerService. In this respect, the task of AdaptationReasonerService is minimal, when the reasoning approach presented in this work is applied.

Adaptation controller

An Adaptation Controller is meant to support pluggable heuristics and distributed reasoning. It is always represented on a local node and it receives events (such as context changes, application launch/shutdown and changes to the set of plans for one application), makes the decision whether an adaptation should be triggered, and, if yes, delegates this task to the Adaptation Reasoner component. The results are then passed to the Configuration Controller in order to start the reconfiguration of the applications.

Adaptation reasoner

An Adaptation Reasoner enables the delegation of adaptation to several nodes. It forwards the requests from the Adaptation Controller to possibly multiple Adaptation Reasoner Services (e.g. local and remote ones) and hands the reasoning result back to the Adaptation Controller. It should provide a fallback mechanism for the case that a remote reasoner is not available. An internal mechanism has to call another remote reasoner or simply use the local reasoner so that the applications are not impacted due to the failure of a remote reasoner. The Adaptation Reasoner is deployed locally.

Adaptation reasoner service

In MUSIC, an Adaptation Reasoner Service is intended to do the actual reasoning. The reasoner service might be exported in order to be used as a remote reasoner. A remote Adaptation Reasoner Service is accessed as a service in the SOA sense. The services are deployed as separate bundles and there might be multiple Adaptation Reasoner Services available on a node at the same time. For our work, the actual reasoning is done by Template Builder.

Configuration controller

The Configuration Controller has to support pluggable heuristics and handle a distributed configuration. It receives the configuration templates from the Adaptation Controller and delegates the realization of such templates to the Configuration Planner and the Configuration Executor. It must be able to handle multiple Configuration Executors (e.g. local and remote). It is always deployed locally.

Configuration planner

The Configuration Planner determines the sequence of steps needed for reconfiguration and creates batches of configuration steps that optimize the configuration. It takes into account application-specific constraints and may use application-specific heuristics during this process. It can be deployed on local and remote nodes.

Configuration executor

The Configuration Executor receives the configuration batches created by the Configuration Planner from the Configuration Controller and executes them. It invokes the Kernel interfaces to deploy the new configuration. For services in SOA sense, it will select the appropriate Service Factory for the service plans depending on the communication protocol.

The performance for configuration is improved in the distributed scenario by handling batches of configuration tasks instead of single steps at a time. The configuration steps need to be synchronized (e.g., in collaboration with other Configuration Executors or with the Configuration Controller). It is deployed on each node to execute the batches for that specific node.

Behavior

The behaviors implemented by the Adaptation Middleware include starting an application, stopping an application, reacting to context changes, reacting to plan changes and the adaptation process. In the MUSIC deliverable D4.2 [60] and D4.3 (see footnote 9) such behaviors are explained using appropriate sequence diagrams. However, compared to MUSIC, in this work, we have updated the adaptation reasoning process and the corresponding behavior is explained using the sequence diagram of Figure 17.

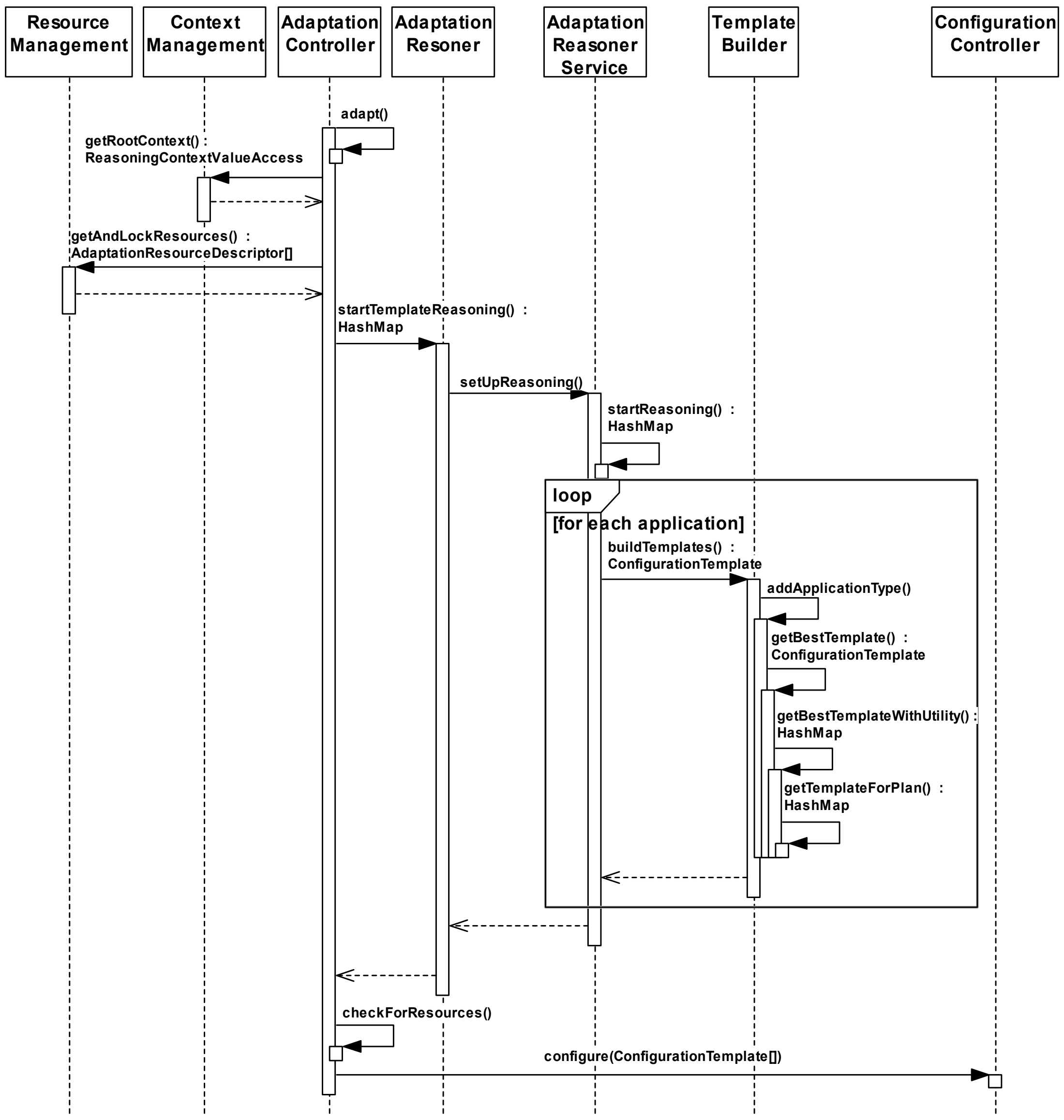

Resource Management
Context Management
Adaptation Controller
Adaptation Resoner
Adaptation Reasoner Service
Template Builder
Configuration Controller
adapt()
getRootContext() : ReasoningContextValueAccess
getAndLockResources() : AdaptationResourceDescriptor[]
startTemplateReasoning() : HashMap
setUpReasoning()
startReasoning() : HashMap
loop
[for each application]
buildTemplates() : ConfigurationTemplate
addApplicationType()
getBestTemplate() : ConfigurationTemplate
getBestTemplateWithUtility() : HashMap
getTemplateForPlan() : HashMap
checkForResources()
configure(ConfigurationTemplate[])

Figure 17: Sequence diagram for the adapt operation within the Adaptation Middleware

This diagram in Figure 17 shows the sequence of steps that occur during the adaptation reasoning process. The adapt() operation is called after a context change, and when an application is started or stopped. The adaptation controller first gets hold of the context information that it uses to reason about the adaptation. Afterwards, information about resources is retrieved from the resource manager and relevant resources are locked for use by the components of the reconfigured application.

Adaptation reasoner is asked to start reasoning over the available templates. It delegates the reasoning task to the adaptation reasoner service. For each of the applications, it asks the template builder for building template. In MUSIC, this returns an iterator to iterate over all the possible variants of the application. However, in U-MUSIC, we perform the actual adaptation reasoning task at the template builder so that it returns only a single template (best-fit) to the adaptation reasoner service.

The template builder adds the type of the application for reasoning and from the repository retrieves (recursively) all the plans in the variability model of the application. The getBestTemplate() method finds the best-fitting configuration template. This method calls the getBestTemplateWithUtility() method to get the all the best-fitting templates[13] with the associated utility for each of the plans. After receiving all these best-fitting templates, the getBestTemplate() method compares their utilities to select the best configuration. The getBestTemplateWithUtility() method uses the getTemplateForPlan() method to retrieve the best-fitting plan variant for each plan. In the case of an atomic or service plan, the comparison is straight-forward. However, a composition plan leads to recursively calling the getBestTemplateForUtility() method for each of its component types in the composition.

The selected variant is checked against resource requirements (and architectural constraints), which may again select another variant that satisfy those constraints (see sections 5.3.2 and 5.3.3). If this variant is different from the running one, a reconfiguration of the application will occur.

Extensions to MUSIC

- The TemplateBuilder Class is enhanced greatly to support the adaptation reasoning process. The methods buildTemplates() and loadComponentType() are updated, and getBestTemplate(), getBestTemplateWithUtility() and getTemplateForPlan() are added. The nested Classes PlanTypeIterator, AtomicPlanItertor, CompositionPlanIterator and NodeIterator are no longer needed and removed.

- The ConfigurationPlanner Class is updated by using plan name instead of component type name in using the createTemplateMap() method. The reason is, the component type name can not be retrieved from the plan, since the getComponentType() method is removed in the information model.

- The ConfigurationTemplate Class is updated by removing the evaluateForRole() method.

[13] Each plan has a number of plan variants. The best fitting plan variant is used to create the configuration template.

- The configure() method of the IConfigurationController interface is updated. It accepts a parameter of type HashMap instead of an array of configuration templates.
- Since adaptation reasoning is basically done in the TemplateBuilder Class, AdaptationReasoner and AdaptationReasonerService may be completely removed. However, these Classes still exist to ensure consistency with the rest of the middleware.

6.1.4 Repository

The Repository is provided by the Kernel component. The bundle registration process is updated in this work, requiring few corresponding updates in the Repository. However, these are minimal and do not require any architectural changes.

6.2 Middleware Implementation

Like the architecture, the details of the middleware implementation are out of the scope of this document. They are described in details in MUSIC Work Package 5 deliverables [2]. However, in the following we describe how two main aspects of this work are implemented updating the MUSIC middleware:

1) Supporting the unanticipated adaptation by runtime creation of the variability model through matching of discovered bundle artifacts
2) The adaptation reasoning process

All updates are made on the MUSIC middleware v0.2.2 released on 17.02.2009 [2]. Some of the updated source code is presented in Appendix A .

6.2.1 Runtime Creation of the Variability Model

The creation of the variability model at runtime involves the following steps:

1. **Installation of a bundle:** A U-MUSIC bundle can be deployed at anytime. Upon deployment, the bundle is installed from the location of the bundle, where the bundle location is specified through a URL. An OSGi service component named Component Context is used by a component instance to interact with its execution context including locating services by reference names. The bundle is installed and started.
2. **Installation of bundle artifacts:** The next step is to install all the bundle artifacts. From the bundle, all the plans, component types and application types are retrieved, and each of them is added in the corresponding repository using appropriate methods. The sequence of installation is flexible. For example, application types or component types can be installed before installing plans. This facilitates the runtime matching of new plans and types with existing artifacts.
    a. **Adding plans to the repository:** While adding a particular plan to the repository all its matched component types and application types are retrieved. The plan is registered against the type name of each of them. In the case, when no matching type is present, the plan is still registered, against a default String instead of a type name. The current implementation of the matching process between a plan and the types

only takes care of the set of functionalities to decide on the matching between a plan and a type. Functionalities of the plan are retrieved and then from the component type repository each of the type is checked, whether its functionalities can be provided by this plan or not. In the current implementation, we have used the string matching technique for types and plans. In connection with the theoretical development, the implementation needs to be updated to ensure the use of the functionality ontology to support imprecise matching, especially when functionalities are similar instead of being exactly the same. Also, it must be ensured that the expected interfaces also match. The matching of plans with application types is done in a similar way, where instead of component types, application types are retrieved from the application type repository and then compared their functionalities with that of the plan.

b. **Adding application types to the repository:** During adding an application type to the repository, the plan repository is first updated with matching plans. The updating of plan repository employs similar technique like the matching of plans with component types. The same method can be used for both component type and application type, because the later is a specialization of the former. All the plans are retrieved from the repository and matched with the component/application type. If it matches, the plan is registered against this type. After updating the plan repository, the application status is set to STOPPED and it is registered in the application type repository.

c. **Adding component types to the repository:** Adding component types to the component type repository is done similarly; however, without setting the status property.

6.2.2 Adaptation Reasoning

The adaptation reasoning process starts in the AdaptationController and the task is delegated to the AdaptationReasoner, which uses the AdaptationReasonerService to perform the adaptation reasoning. In MUSIC, the AdaptationReasonerService uses TemplateBuilder to build templates corresponding to all possible application variants and then reason about the best fitting variant applying the reasoning algorithm. However, in this work, the adaptation reasoning algorithm is updated and it is included in the TemplateBuilder so that it always returns only a single template corresponding to the chosen application variant. The AdaptationReasonerService simply provides this template to the AdaptationController to reconfigure the application. Therefore, in effect, adaptation reasoning is done during building templates in the TemplateBuilder. The reasoning process involves the following steps:

1. **Initiation of building templates:** The TemplateBuilder Class uses the buildTemplates() method as the starting point for retrieving the best-fit template. By adding an application type all the plans for that type are retrieved from the plan repository with the help of a resolver. The approach is very simple: keep a map from component type to all plans for the type. If the plan is an instance of a CompositionPlan, all the plans corresponding to the types present in the composition are also recursively retrieved. Context

dependencies for the plan are also filled. In order to ensure that the root component is deployed on the MASTER node, an enumerator is used. Afterwards, the getBestTemplate() method is called to find the template corresponding to the best fitting variant of the application.

2. **Retrieval of the best template:** The getBestTemplate() method actually uses the getBestTemplateWithUtility() method to find the best configuration template. The later provides a HashMap with the template and the corresponding utility value, while this method retrieves the template from the HashMap and returns it. If no template is found, null is returned. The getBestTemplateWithUtility() method retrieves all the plans for a particular type and for each of the plans, creates a template along with its utility using the getTemplateForPlan() method. After retrieving these plans, the one with the highest utility is returned as the best fitting template.

3. **Retrieval of the best template for each plan:** The getTemplateForPlan() does the bulk of the calculation, as it retrieves the best template for each of the plans, choosing from all the possibilities corresponding to the variants of the plan. If the plan is an atomic realization plan or service plan, utilities of all its plan variants are calculated and the variant providing the highest utility is chosen among the existing variants for this particular plan. A configuration template is created using this plan variant. If the plan is an instance of a Composition plan, the utility of the composition needs to be calculated from the utilities of its constituent component types. For each of the types in the composition, the getBestTemplateWithUtility() method is recursively called. Child templates are created using the type name (role name in the source code corresponds to a component type name) and the best fit configuration template for it. The utility of the composition is calculated using the weights of part utilities and the values of those part utilities for constituent types. Among the variants of the plan, the one providing the highest utility is chosen. The best template is returned to the AdaptationController for starting the reconfiguration process.

6.2.3 Implementation Status

The current update of the middleware provides a preliminary implementation of the unanticipated adaptation related concepts described in this thesis. The runtime matching process performs only String matching and the functionality ontology is not yet supported. Therefore, at present no support for imprecise matching is provided. As an enhancement to MUSIC, the implementation facilitates using a particular realization plan for a number of different component types; however, the integration of the functionality ontology is a high priority task for supporting unanticipated adaptation.

The implementation of the adaptation reasoning algorithm supports the basic reasoning approach (section 5.3.1). However, it must be enhanced by providing the support for meeting resource constraints (section 5.3.2) and architectural constraints (section 5.3.3) to ensure the selection of a practically feasible application variant.

7 Methodology and Tools

The design and implementation of unanticipated adaptive applications in the envisaged ubiquitous computing environment is certainly a great challenge for application developers. Therefore, it is an important objective of this work to complement the middleware forming the runtime infrastructure of adaptive applications with an elaborate development methodology to support developers.

The development methodology not only provides a step-by-step guideline to specify the application's context dependencies, variability and domain model, but also covers the transformation of models to source code, the deployment of applications on the middleware and the testing and validation of the expected adaptation capabilities.

In support of the methodology, the use of tools aids the developers in different development steps. This chapter introduces those tools as well. Both the methodology and tools are dependent on MUSIC and therefore, in this document, we mostly provide a high level description with highlighting the updates made on MUSIC results [64]. Many of the aspects of the methodology are therefore only briefly introduced for completeness or not described at all when it does not hamper understanding the methodology.

7.1 Model Driven Development Approach

The model-driven development approach adopted in this work follows the principles of the Object Management Group (OMG) Model Driven Architecture (MDA[14]), which is the best-known and most widely used Model Driven Development (MDD) initiative and which functions as a reference architecture for most MDD approaches. According to the MDA,

> "*The MDA approach and the standards that support it allow the same model specifying system functionality to be realized on multiple platforms through auxiliary mapping standards, or through point mappings to specific platforms, and allows different applications to be integrated by explicitly relating their models, enabling integration and interoperability and supporting system evolution as platform technologies come and go". [62]*

In general, the MDA focuses primarily on the functionality and behavior of a software system, rather than on the specific technology based on which it will be implemented[15]. OMG defines the notion of Platform Independent Model (PIM) and Platform Specific Model (PSM). A PIM focuses on the operation of a system while hiding the details necessary for a particular platform; whereas a PSM adds the detail of the use of a specific platform by a system.

[14] Model Driven Architecture (MDA) is a Registered Trademark of the Object Management Group.

[15] However, in this work we are concerned on modelling adaptation capabilities of an application, rather than its functionalities and behavior.

The primary goals of the MDA are portability, interoperability and reusability through architectural separation of concerns. Abstracting out the fundamental precise structure and behavior of a system in the PIM from implementation specific concerns in the PSMs has three important benefits:

1) easier validation of the correctness of the model uncluttered by platform-specific semantics,
2) easier production of implementations on different platforms while conforming to the same essential and precise structure and behavior of the system and
3) clear definition of integration and interoperability across systems in platform-independent terms and then mapping them down to platform specific mechanisms[62].

In addition to providing most of the above benefits offered by the MDA in general, the model driven development approach provided in this work originates from the aim of supporting a number of aspects based on which unanticipated self-adaptation is offered to mobile applications. These aspects comprise the following:

- **Supporting variability:** A conventional Model Driven Development approach consists of creating a Platform Independent Model (PIM) of the software architecture. The PIM can be transformed to Platform Specific Model (PSM) in a number of steps. In contrast to the conventional approach in MDD and MDA, our approach does not mainly focus on the platform independency dimension, but rather on the application variability. A variability model is created at runtime by the middleware; but its constituents - application types, component types and plans describing the realization details of such types - are modeled at design time. In order to support the unanticipated adaptation, such plans and types can be modeled by different developers without prior knowledge about each other. A PIM for variability thus contains any number of component types, application types and/or plans. A model of a type generally contains its interfaces and required functionalities, while that of a plan also contains properties and utility functions, resource requirements, distributions of component types, compositions, references to components etc.

- **Language extensions for modeling context:** As an adoption of the work from the MUSIC project, language extensions and specification means to model context information, context collection, and reasoning mechanisms are also supported. This allows the generation of appropriate source code from these specifications that can be utilized by the applications and the middleware framework. In the same way as for the application variability model, appropriate modeling notation is used to specify the PIMs. A number of tools are developed to perform the transformation to the corresponding PSMs and to source code.

- **Integration of models and ontologies:** Another important motivation behind the proposed approach is the combination of conventional MDD with semantic modeling of context properties, domain knowledge, and user profiles. While the general system behavior and the generation of execution variants is governed by the MDD methodology, the high flexibility of the adaptation to the delivery context as well as the different domains is achieved by the usage of semantic technologies. This is more important for supporting the unanticipated adaptation. For example, a common vocabulary of functionalities with the support of an extensible ontology may facilitate the use of plans for realizing component types developed by different developers.

7.2 Methodology

The work follows the model-driven development paradigm, as introduced in section 7.1. The modeling notation is very much similar to that presented in the MUSIC Deliverable D6.3 [63], while the methodology from the MUSIC project [64] is updated to enhance the support of the unanticipated adaptation.

With the support of appropriate modeling notation, the application developer is enabled to specify the application variability model, context elements and data structures, as well as component functionalities and QoS properties at an abstract and platform-independent level. The source code necessary to publish the adaptation capabilities, context dependencies, variability with regard to external services and properties of the application to the U-MUSIC middleware is automatically generated by model transformations. This eases the development of adaptive applications to a large extent, as the application developer can concentrate on the application adaptation model and is not confronted with implementation details.

However, the modeling support and the code generation facilities provided by U-MUSIC focus on the adaptation capabilities of an application. A general MDD approach for the functional parts of the applications is beyond the scope of U-MUSIC, since it is intended to provide support for applications of a large variety of domains. Therefore, development approaches and code generation for the functional parts (components) of the application are not addressed in the methodology.

The methodology is viewed as a step-by-step guideline for the application developers, as depicted in the overview of Figure 18. The methodology comprises five main tasks:

- Analysis
- Modeling
- Model Transformation
- Packaging and Deployment
- Testing and Validation

Each of these tasks includes several sub-tasks and therefore, a completely step-by-step approach can be defined to describe in details what an application developer needs to do. Those steps are extensively described in MUSIC deliverables D6.2 [64] and D6.4 [65]. This thesis follows the same methodology, while updating that to provide an enhanced support of the unanticipated adaptation. Therefore, here we will mainly highlight the updates, while each step will be briefly introduced for the sake of completeness.

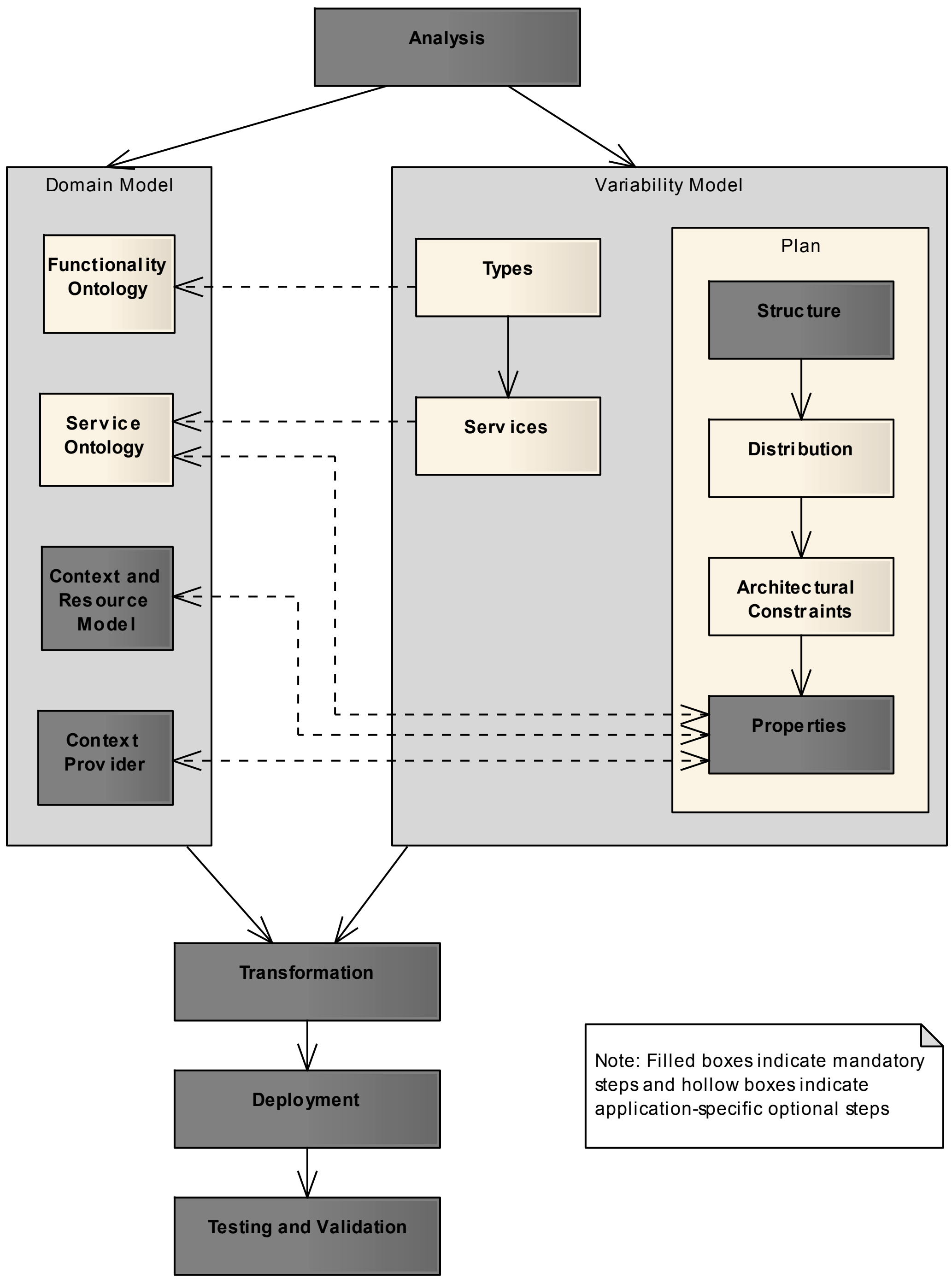


Figure 18: Overview of the methodology

7.2.1 Analysis

The Analysis phase is the starting point for developing the application. The support for the unanticipated adaptation requires an insightful analysis of what the application would do along with the possible context and resource dependencies. The first step is to find the functionalities that are supposed to be performed by the application or certain components. Such functionalities may be provided by components developed by the same developer or an independent developer or by external services. Such components may be deployed along with the application and meta-information (plan) about the realization, or they may be deployed any time later or discovered at runtime. A developer performs a requirement analysis resulting in a list of functionalities of the application and part-functionalities that are influenced by changes in the execution context.

In order to be aware of possible resource and context dependencies, a rough understanding of the execution environment is required. For this purpose the MUSIC ontology [64] [65] includes a collection of resources and context elements which is intended to provide an elaborate set of example dependencies and helps the application developer to establish an initial list of resource and context dependencies. This initial list will be leveraged to specify the resource and context model as part of the domain model later on. Furthermore, the application developer has to be aware of different nodes constituting the distributed execution environment and has to get an overview of potential external services available in the execution environment. However, for an unanticipated adaptation, it is often impossible to get a complete picture of the nodes and services that will be available in the adaptation domain during the runtime of the application.

Based on the above considerations, the analysis phase can be divided into four sub-tasks:

- Identification of functionalities
- Identification of potential context and resource dependencies
- Identification of potential node types
- Identification of potential services

Identification of functionalities

In this work, we support the possibility of developing different components of the application separately, possibly by separate developers. Therefore, some developers may specify 'what' their application or component is supposed to do, while some others may specify the need for components that will perform some tasks. Functionalities are defined as which task a particular component and/or application will perform. In the methodology, we do not consider how such functionalities are implemented by certain components and/or applications; rather we focus on the functionalities to be supported by an application or a component. We adopt the idea of composite components and an application can, in general, be viewed as a composition of different components and services. Therefore, a functionality itself can be realized by a number of part functionalities.

To illustrate the identification of functionalities, we consider the scenario presented in section 1.2, where we can identify a number of different functionalities as listed in Table 2.

Table 2: List of functionalities for the scenario of section 1.2

Functionality	**Refers to (Application/ Component/Service)**	**Description**
Assist traveler	Application	Top level functionality of the UnanticipatedTravelAssistant application. The functionality can include any number of part-functionalities that collectively helps the traveler.
Navigate	Component	Navigation functionality
Create itinerary	Component	Aid creation of travel plans
Process image	Component	Analyze and edit collected image
Process video	Component	Analyze and edit/manage collected video
Take picture	Component	Take images with a camera
Take video	Component/Service	Take video with a camera
Record video stream	Component/Service	Record video from own device or from a service
View map	Component/Service	Map viewing on a screen
Download map	Component/Service	Downloading map
Plan route	Component/Service	Route planner
User interface	Component/Service	User interface to communicate with the device
Heads up display	Component/Service	A transparent display
Mobile device display	Component	Display of the mobile device
Voice command	Component	Voice command facility of the mobile device
Text-based UI	Component/Service	Text input through keyboard
Touch screen UI	Component/Service	Touch screen input
Play music	Component	Play audio
Buy ticket	Component	Ticket buying activity (for entry to the

		garden and car parking)
Sell ticket	Service	Ticket selling activity by the service provider
Verify ticket	Service	Ticket verification
Search image	Component/Service	Searching for images based on meta-information
Sort image	Component/Service	Sorting images according to their quality
Select image	Component/Service	Select a number of images
Upload image	Component	Upload selected images
Control radio	Component	Selecting radio channel
Find ad	Service	Finding advertisements

Such a list of functionalities is used during the modeling phase to create the variability model. Some of the functionalities may be considered as core functionalities, while some others may be supported on demand.

Identification of potential context and resource dependencies

In order to develop an application that is able to retain a high-quality of service in a dynamically changing mobile and ubiquitous computing environment, the application developer has to be aware, as much as possible, which characteristics of the execution environment affect the operability and the perceived quality of service of the application. Therefore, after the general idea and a rough picture of the application have been established, the first step is to identify an initial set of critical context and resource dependencies. It provides first hints, which adaptations have to be covered and which high-level configurations or modes of the application should be provided. The list of context and resource dependencies will be refined later on when creating the application variability model, i.e. when more insights of the application's configurations and its constituting components are available.

It is not easy to get aware of all the potentially influencing resource and context characteristics of the execution environment, especially when dealing with the unanticipated adaptation. The MUSIC project provides a list of potential context and resource dependencies. The list is derived from the resource and context model as part of the MUSIC Domain Model in terms of the MUSIC ontology, that serves as the basis for all the pilot service developments in MUSIC. As the scenarios for the pilots cover quite different application domains, the list can be expected to be comprehensive enough to serve as a good baseline for the development of adaptive applications in general. The application developer just has to go through the list and mark the context and resource dependencies relevant for the application to be developed. However, when a particular application depends on a certain context entity or resource that is not listed already, the developer can extend the Ontology. As a result of this task, an initial list of critical context and resource dependencies is available that guides the developer when specifying the application variability model.

Before identifying an initial list of context and resource dependencies, first we have to define the different types of context and resource entities. In general, a context dependency can refer to a certain entity (e.g. user) and to a scope of a certain entity (e.g. the location of the user). Resource dependencies exist with regard to a resource entity (e.g. Memory), and refer to a resource service of the entity (e.g. JVM_MemoryService of the entity Memory) and to a scope of this resource service. MUSIC Ontology [63] provides an initial list of potential context and resource entities and scopes. The application developer should have a look into these classes to get further ideas how to expand this list. For the scenario presented in section 1.2, the developer of the UnanticipatedTravelAssistant application would identify the context and resource dependencies as depicted in Table 3.

Table 3: List of context and resource dependencies of the UnanticipatedTravelAssistant application

Resource	Battery, CPU, Memory, Network, Screen, Speaker
Person	User, scope: Location
	User, scope: Environment (light)
	User, scope: UserPreferences (profile)
Device	Car, scope: Location (navigation component foresees car)

The dependency of the application to context and resources are used to model the QoS properties of different realizations of the application and its components. Also, the runtime values of different context and resource properties are used to evaluate the fitness of certain variants of the application during the adaptation reasoning process.

In this work, we have presented an adaptation reasoning approach that focuses on the QoS properties of individual components and therefore, a developer needs to focus only on the context and resource dependencies of the components he is developing. This is particularly helpful in the case of the unanticipated adaptation, because the developers do not need to bother much about such dependencies of components developed by others. However, the representation of context and resource values plays an important role and hence the ontology eases the integration task.

Identification of potential node types

We address a distributed execution environment and adaptations influencing the distribution of components over different nodes. In order to support the application developer with the identification of possible adaptations regarding distribution, a model of the execution environment in terms of its constituting nodes becomes helpful. The execution environment is modeled as a set of communicating node types. With the type concept we abstract away from concrete nodes. This introduces further variability, as the node types can be instantiated through a number of concrete nodes realizing a particular type. In general, the execution environment is modeled as a hierarchy of node types, which means a node type can contain several other child node types. A node type belongs to the U-MUSIC adaptation domain, if it is running an instance of the U-MUSIC middleware. Each node type in the U-MUSIC adaptation domain can host components or component compositions that can be used to compose the application to be developed. Identification of external node types, not running an instance of the U-

MUSIC middleware and therefore, not a direct part of the U-MUSIC adaptation domain, is also important as such node types may host third-party services that can be integrated into the application.

A node type can contain several other node types. In this case, the container node type is considered as (part of) an execution environment, which consists of the included node types. With such a concept, node types can be grouped together, e.g. nodes that are running the U-MUSIC middleware versus other nodes that do not take part in the adaptation process. This allows distinguishing the U-MUSIC adaptation domain from external nodes that may host external services.

In the case of the unanticipated adaptation, the exact nodes or type of nodes that will appear in the middleware domain can not be always foreseen during the analysis phase or at design time. External nodes (outside the middleware domain) may be considered as service providing nodes and the type of such nodes may be identified analyzing the expected functionalities of the application and probable service providers that may offer them. For nodes that are part of the U-MUSIC domain, we define two node types at the moment: MASTER and SLAVE, which dictate the adaptation reasoning. With this categorization, we consider all U-MUSIC nodes which are not equal to the user's mobile device as SLAVE. As for example, in the scenario of section 1.2 the device of Thomas is considered as a MASTER with respect to the UnanticipatedTravelAssistant application, while that of Stephan is considered as of type SLAVE. If the car computer is running an instance of the U-MUSIC middleware, it can be also considered as a SLAVE. Otherwise, it will just be a service provider node like the coffee machine at the petrol station (Scene 2, section 1.2.2). In the scenario, the World Wide Web is another node type that provides the ticket buying service or announces the availability of nearby restaurants.

The identification of such nodes needs not to be rigorous, especially because all possible node types may not be foreseen during this analysis.

Identification of potential services

Ubiquitous computing environments are characterized by dynamically discoverable and accessible services. One main aspect of dynamic adaptation is to use those services to realize (part-) functionalities of the application. Although not primarily designed for U-MUSIC, such external services still represent composable entities that may be integrated into the application to enhance its functionalities or to improve the quality of service perceived by the user.

The identification of potential services depends upon the list of functionalities (Table 2) prepared in the 'Identification of functionalities' sub-task. Some of the functionalities may be completely bound to be provided by the developer himself or other U-MUSIC components. However, some functionalities may be provided either by a U-MUSIC component or an external service, while the realization of some functionalities may be solely dependent on external services. When designing the application variability model for identifying the component types that are realizable through external services, such component types have to be semantically annotated.

Extensions to MUSIC

The support for unanticipated adaptation has triggered the following updates to the analysis phase of the MUSIC methodology:

- The functionality concept is introduced for unanticipated adaptation. Therefore, the task 'Identification of functionalities' is also added.
- The scopes of the other three tasks become limited in the case of unanticipated adaptation. It is not usually possible to identify the context and resource dependencies, when the components will be provided by other developers. The deployment of components to specific nodes is also not possible to foresee at this phase. However, currently the node types are divided in two categories only and therefore, all nodes, except the one that the user of the application holds, can be considered of the SLAVE type.

7.2.2 Domain Model

The Domain Model is used to define the execution environment in terms of context, resources and available services in an unambiguous way. Therefore, the execution environment is described through concepts defined in an ontology. For the basic modeling and structuring concepts of the ontology we refer to the Deliverable D6.3 [63] of the MUSIC project. However, in order to understand the following paragraphs, it is important that the Domain Model is formed of the MUSIC top-level ontology covering general knowledge applicable to a wide range of adaptive applications and of one or several sub-ontologies covering concepts more specific to the actual domain of the application to be developed.

In addition to the Domain model presented in MUSIC, we need to define a Functionality Ontology, similarly to the Service Ontology so that U-MUSIC nodes can be discovered and used in an unanticipated way. In general, defining and updating the MUSIC Domain Model of the application incorporates the following sub-tasks.

- Defining and updating the Context and Resource Ontology
- Checking the availability of context providers
- Defining the Service Ontology
- Defining the Functionality Ontology

The first three of those subtasks are described in details in MUSIC deliverables D6.2 [64] and D6.4 [65]. However, here we briefly introduce them for completeness.

Defining and updating the context and resource ontology

Context-aware and adaptive applications depend on the QoS properties of the context and resources. In the approach presented in this thesis, based on the QoS properties required by the application (components, in general) and that provided by the execution context, an application variant, which provides the best utility, is chosen. It has to ensure that all the properties can be derived from context or resource information available in the system and all referred concepts are based on the vocabulary defined through the MUSIC ontology. This also means, that the Domain Model for the application reflects the execution environment in an appropriate manner and to an appropriate level of detail.

In the analysis phase an initial list of context and resource dependencies is derived based on the context and resource model that is provided through the MUSIC Ontology, i.e. the top-level ontology as the basis for all applications and the sub-ontology (ontologies) corresponding to the actual application domain. The initial list of context and resources can be updated in later phases, when the variability model (section 7.2.3) requires the specification of properties of context and/or resource entities that are already not included in the list or in the Ontology.

The MUSIC context and resource meta-model are presented in the MUSIC deliverable D2.2 [12]. Each Context entity is characterized by its value, scope, representation, source and some meta-data. Each value corresponds to a dimension having a certain representation. Resources are modeled in the same way, with the extension that resources can provide services, which can be utilized by the application and/or the middleware platform.

With regard to the Domain Model for the application all the referred context entities, resource entities and its services and also the scopes and their representations have to be available in the ontology. Therefore, the application developer has to check, if the corresponding class is available in the MUSIC top ontology. If it is not available, then also the sub-ontology has to be checked. In the case, that it is also not available in the sub-ontology, the sub-ontology has to be updated with the missing class or classes. Before updating the ontology it has to be checked if a context sensor or resource sensor can be incorporated (already available or to be developed) to provide the expected information in the expected representation. If not, then the application has to be refined to be not dependent on such kind of context or resource information.

Every context or resource element refers to a certain entity type in the MUSIC ontology. Figure 19 illustrates the current hierarchy of entity types.

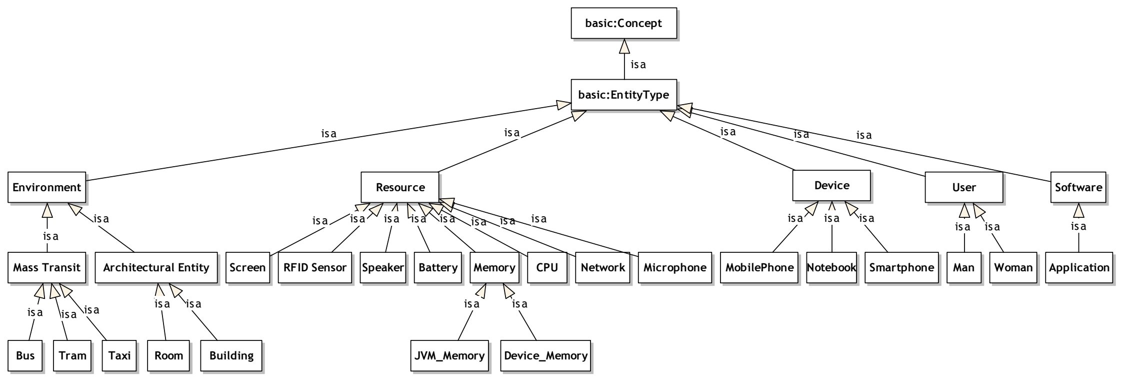


Figure 19: Context and resource entity types

Furthermore each context or resource element refers to a scope and its representation in the MUSIC ontology. Figure 20 shows an example of such a scope and its representation. In this example the scope BatteryServiceDescription is shown, which is used to describe the BatteryService of the Battery resource entity. Furthermore the figure shows the different properties of a possible representation[16] of this scope.

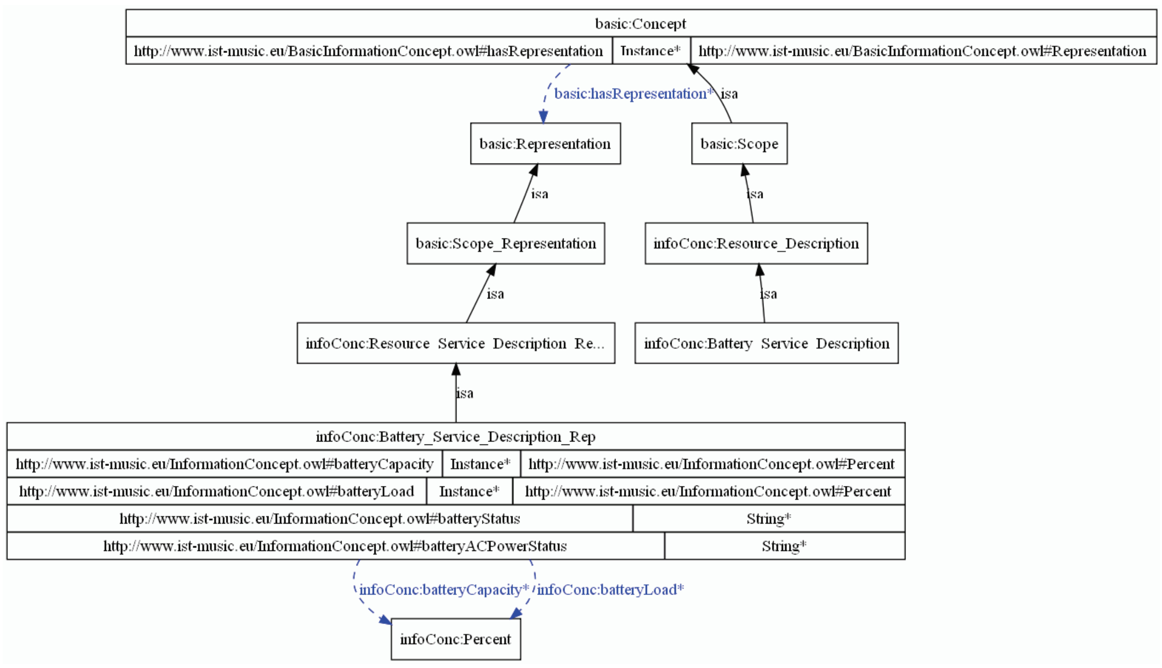

basic:Concept
http://www.ist-music.eu/BasicInformationConcept.owl#hasRepresentation
Instance*
http://www.ist-music.eu/BasicInformationConcept.owl#Representation
basic:hasRepresentation*
isa
basic:Representation
basic:Scope
isa
isa
basic:Scope_Representation
infoConc:Resource_Description
isa
isa
infoConc:Resource Service Description Re...
infoConc:Battery Service Description
isa
infoConc:Battery_Service_Description_Rep
http://www.ist-music.eu/InformationConcept.owl#batteryCapacity
Instance*
http://www.ist-music.eu/InformationConcept.owl#Percent
http://www.ist-music.eu/InformationConcept.owl#batteryLoad
Instance*
http://www.ist-music.eu/InformationConcept.owl#Percent
http://www.ist-music.eu/InformationConcept.owl#batteryStatus
String*
http://www.ist-music.eu/InformationConcept.owl#batteryACPowerStatus
String*
infoConc:batteryCapacity*
infoConc:batteryLoad*
infoConc:Percent

Figure 20: Ontology example for scope and representation

Checking the availability of context providers

The specifications of the Context and Resource models along with the properties in the variability model is only the half of the job, because the context and resource information has to be retrieved by appropriate sensors and reasoners at runtime. Such context providers may be provided by individual application developers and may be shared by others. However, if they can not be expected to be installed on the system or to be dynamically discovered at runtime, an appropriate context provider has to be developed.

First of all, the nodes of the adaptation domain have to be inspected if they provide the appropriate context sensors. The appropriateness of a context sensor can be judged by checking its associated meta-data that specify which context scope in which representation is provided. The characterized entity (and possibly the resource service) is either explicitly defined in the meta-data too, or implicitly given through the device/platform that runs the context provider. If no node type of the adaptation domain can be expected to run an appropriate context provider, then the node (types) external to the adaptation domain but available in the execution environment are checked in the same way as U-MUSIC nodes.

If no appropriate context provider is available, then it has to be decided if the development of a new context provider is feasible or not. The decision mainly depends on aspects like the necessary effort to get the information from a hardware component, if hardware support for an appropriate context sensor is available at all or how much

[16] A scope or in general a concept could have several representations. Therefore, we say “possible representation”.

effort has to be spent to derive the information from already available information. If it is not feasible to develop a new context provider, then it has to be checked if alternative properties resulting in different needs for context and resource information can be used, to achieve a similar adaptation decision. If this is also not possible, then the general idea of the application has to be reconsidered.

This is apparently a big challenge for the unanticipated adaptation, because at design time, the nodes that may appear or disappear in a completely unanticipated manner can not be foreseen. Therefore, it is also unknown, which sensors or reasoners might be available at runtime. One way to solve this problem is to focus on the need of individual components. When a developer is providing a number of components, then he might also provide the sensors and reasoners to retrieve context and resource information specified by the properties of the realization plan of those components. However, this might result in superfluous providers, because a context provider may serve a multiple number of context clients. On the positive side, it improves the context quality and the dynamic discovery for context provisioning becomes a more realistic option.

The MUSIC project offers the support for developing context plug-ins to be used as context providers (sensors and reasoners). It also follows the model driven development paradigm. For details, please refer to the Deliverable D6.3 [63].

Defining the service ontology

The U-MUSIC middleware allows an application to utilize external services and to integrate it into the application's component composition. For this purpose an external service is considered as an alternative realization for a component type. The integration into the application's composition is realized through a service proxy component that acts as a local representative of the service and establishes the binding to it. In order to facilitate dynamic service discovery, a component type (or, more precisely speaking, its port type) is associated with a service description that specifies the information needed to discover a service; e.g., a service classification, service negotiation protocol, property types etc. The service classification refers to a concept in a semantic taxonomy of service functionalities defined in the MUSIC Service Ontology. From that perspective, a service classification has the similarity with the functionality of component types. However, a component type is characterized by any number of functionalities, while a single service classification refers to a particular type of service. Currently, only the service classification is considered during service discovery and matching. It is assumed for the moment, that an external service with the expected service classification provides the appropriate functionality, ports and interfaces to be incorporated into the application's component composition. It has to be ensured that in a service description the service classification refers to a semantic concept in the MUSIC Service Ontology.

A list of possible services is prepared in the analysis phase. The developer checks if service classifications for them are available in the MUSIC Service Ontology, either in the MUSIC top-level ontology or in one of the applied MUSIC sub-ontology. If the concept is not available, the taxonomy of service functionalities in the sub-ontology is enhanced with the new concept. As the result of this sub-task, an ontology is available that facilitates semantic service discovery and matching on a very basic level.

Figure 21 shows the hierarchy of the service classification in the MUSIC ontology. Currently this classification only contains resource services. An extended hierarchy has to be defined in the future and therefore existing taxonomies of service categories like NAICS [66] and UNSPSC [67] have to be taken into account.

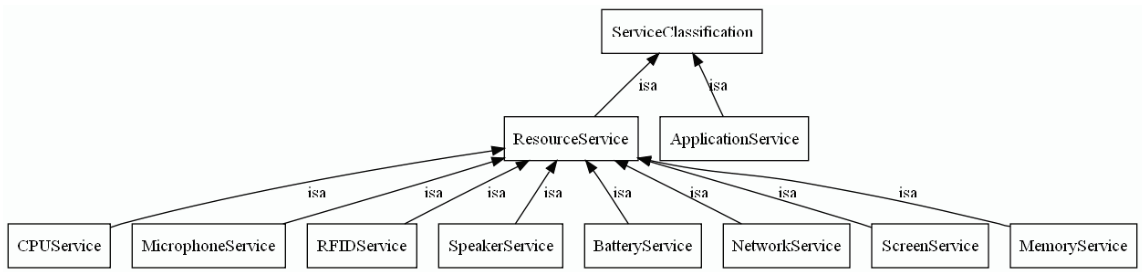


Figure 21: Service classification

The support for using external service is completely adopted from the MUSIC project.

Defining the functionality ontology

The need for the Functionality Ontology has similar grounds like the need for the Service Ontology. The difference is that it will facilitate the inter-operability and usage of U-MUSIC components developed by different application developers, while service providers can be completely ignorant of U-MUSIC at all. It might be a good idea to unify the functionality ontology with the service ontology. However, currently service classification focuses only on resource services, as shown in Figure 21. On the other hand, we need to distinguish among different kinds of functionalities, namely core functionalities, optional functionalities and ad hoc functionalities. Therefore, we define a separate ontology for expressing the relationship among functionalities. In the future, it might be convenient to use a unified ontology for defining both service classifications and functionalities. The Functionality Ontology is presented in Figure 22.

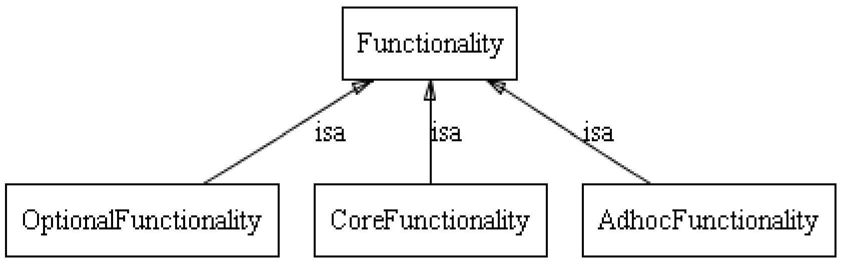


Figure 22: U-MUSIC functionality ontology

A component type defines a number of functionalities that are supposed to be provided by a component realizing that type. The useful components are identified by the meta-information in the component plans that promise to provide the functionalities. In MUSIC, the term ‘type name’ is used to establish such correspondences. However, it is unlikely that two different developers, who are completely unaware of each other’s work, would use the same term as the type name. Therefore, we use the ontology to provide a common vocabulary to independent developers. MUSIC approach also needs a static binding between a component type and a realization plan and therefore, a plan can be used to realize only a particular component type. In practice, such limitations should be avoided, because it can be possible that a component (plan) can offer a set of functionalities, where any component type requiring a sub-set of those functionalities can be easily satisfied with its needs. Another aspect is, like the service matching case, sometimes it is possible that a component is roughly providing the functionalities required by a component type. This is particularly useful for the unanticipated adaptation in order to maximize the benefit from available components and devices.

Each developer has to extend the functionality ontology (Figure 22) describing the functionalities that are relevant for his applications and/or components. During the matching process, the functionalities (as well as the interfaces and ports, though we have not provided that support in the current implementation of the middleware) are also to be taken into account. The extension to the Functionality Ontology depends on the purpose of the application. Usually, applications designed for similar purposes would provide similar set of functionalities.

Extensions to MUSIC

The support for unanticipated adaptation has triggered the following main updates to the domain modeling phase of the MUSIC methodology:

- The functionality ontology is added to provide a common vocabulary for independent U-MUSIC component developers. This facilitates the inter-operability among different developers and thus components developed by one can be used by others.

7.2.3 Variability Model

Variability model is the basis for creating application variants at runtime. The basic idea is simple. Component types are considered as variation points, where each component type can have a number of different realizations. The details of a particular realization are described in a plan. In the case when the component type is realized by a single component or a service, they have corresponding atomic or service plans. But when the functionalities of a component type can be realized by a composition of components; e.g., each component realizing some part functionalities, the plan (composition plan) contains a composition of component types, where each type can further have a number of different realizations. Resolving all the variation points, by selecting a particular plan from the alternatives for each of the component types will create a particular application variant.

The variability model created in this work is different from what is supported by MUSIC. In MUSIC, though plans for a particular type can be developed separately, each of the developers must have the type information. However, this is not possible for the unanticipated adaptation, because the developers are considered as independent and it is unlikely that they would use the same type name. In MUSIC, the link between a plan and a component type is established statically at design time by the ‘type name’. In this work, we remove such static link between a type and a plan; rather they are matched dynamically at runtime using more fine-grained meta-information like the functionalities a type requires and a plan provides, along with interfaces, properties etc.

Such considerations allow separating the development of types from plans and therefore, individual deployment units (bundles) from different users may contain only types, only plans or both.

Modeling of Types

The variability modeling starts with the modeling of application types and component types. In the case of the unanticipated adaptation, the task is quite simple, because everything the developer needs to do is to express his needs of functionalities that he expects the application to perform. In general an application can have a number of functionalities; however, a top level functionality may even suffice as shown in Figure 23, corresponding to the UnanticipatedTravelAssistant application used in the scenario of section 1.2.

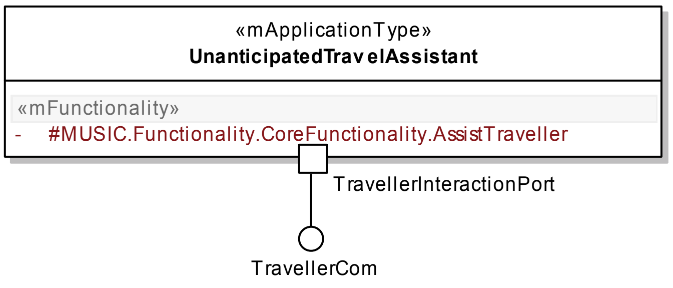

«mApplicationType»
UnanticipatedTrav elAssistant
«mFunctionality»
- #MUSIC.Functionality.CoreFunctionality.AssistTraveller
TravellerInteractionPort
TravellerCom

Figure 23: Modeling of application type

The purpose of the application may be expressed using the functionality 'assist traveler' (AssistTraveller in Figure 23), while this functionality can have other part functionalities like 'process image', 'create itinerary', 'view maps' etc. (see Table 2) that are expected to be provided by individual components.

Component types are modeled in a similar way as presented in Figure 24.

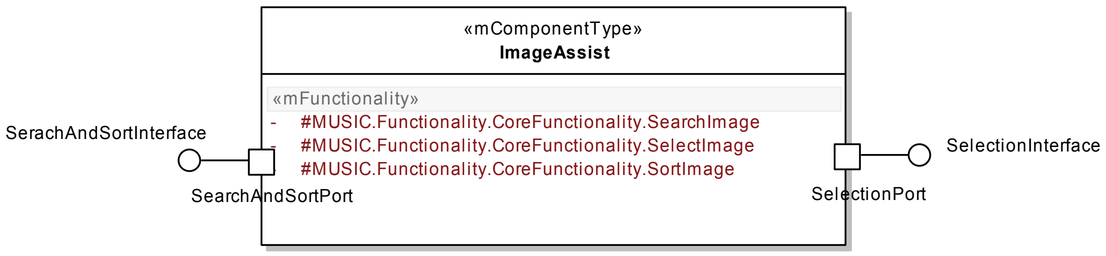

«mComponentType»
ImageAssist
«mFunctionality»
- #MUSIC.Functionality.CoreFunctionality.SearchImage
- #MUSIC.Functionality.CoreFunctionality.SelectImage
#MUSIC.Functionality.CoreFunctionality.SortImage
SerachAndSortInterface
SearchAndSortPort
SelectionInterface
SelectionPort

Figure 24: Modeling of component types

The ImageAssist component type expects a component (atomic or composite) to realize three functionalities: searchImage, sortImage and selectImage. The component type designer does not need to care much about how these functionalities will be realized; for example, some component provider may provide all these functionalities through a single atomic component, while some may use a composition of components, one realizing the search and sort functions, while the other providing the select functionality.

One notable difference here with respect to the MUSIC methodology is that the developer does not need to think much about the realizations; they just express 'what' is expected to be done by a realization of the type, not 'how' it will be realized.

Modeling of service needs

To allow the realization of a component type through an external service, the application developer has to mark this component type. For this purpose, a service description is provided in addition to the modeling of interfaces and functionalities for the corresponding component type (see Figure 25). The most important attribute of the

TicketServiceDescription Class, stereotyped as «mServiceDescription», is the service classification, which refers to a concept in the MUSIC Service Ontology (see section 7.2.2) and describes the general functionality with a common vocabulary. Currently, only the service classification is used for service discovery and matching. In addition to the service classification, the service description also contains the property types of the service that are used in service plans and thus contribute to the adaptation reasoning process. A dynamically discovered service is expected to provide information about its properties corresponding to the specified property types.

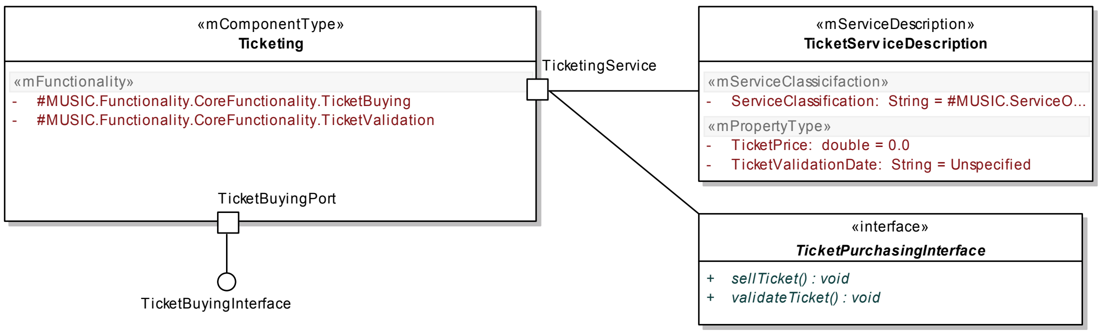

«mComponentType»
Ticketing
«mFunctionality»
- #MUSIC.Functionality.CoreFunctionality.TicketBuying
- #MUSIC.Functionality.CoreFunctionality.TicketValidation
TicketBuyingPort
TicketBuyingInterface
TicketingService
«mServiceDescription»
TicketServiceDescription
«mServiceClassicifaction»
- ServiceClassification: String = #MUSIC.ServiceO...
«mPropertyType»
- TicketPrice: double = 0.0
- TicketValidationDate: String = Unspecified
«interface»
TicketPurchasingInterface
+ sellTicket() : void
+ validateTicket() : void

Figure 25: Modeling of service needs

The prediction of useful services is not completely unanticipated, because the developers can predict which component types might be realized by external services, they can also predict to some extent the type (service classification) of services that fit for this purpose.

Modeling of plan: structure

The modeling of plans is separated form the modeling of types so that the contents of a deployment unit (bundle) may contain only plans or types or both. Each plan is modelled within a separate package, stereotyped as «mAtomicRealization», «mCompositeRealization» and «mServiceRealization» for atomic plan, composition plan and service plan respectively. In the case of an atomic plan, a Class stereotyped as «mComponent» is presented in a Class diagram within the realization package (see Figure 26).

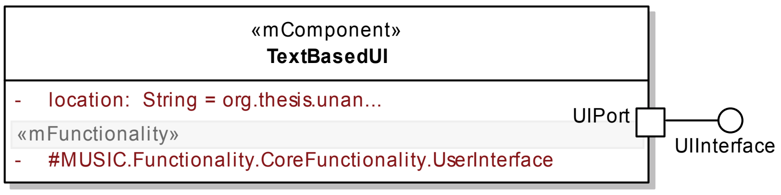

«mComponent»
TextBasedUI
- location: String = org.thesis.unan...
«mFunctionality»
- #MUSIC.Functionality.CoreFunctionality.UserInterface
UIPort
UIInterface

Figure 26: Modeling of the TextBasedUI component

The TextBasedUI component is modeled with the interface it implements along with the ‘userInteface’ functionality. Other types of user interfaces may also be modeled the same way implementing the same interface.

In order to create application variants using the variability model at runtime, a composition plan contains a composite structure consisting of component types instead of components. Therefore, to design a composition plan, all the part functionalities have to be identified, where a number of part functionalities can be realized by a particular component. Such part functionalities are abstracted away by component types within the composite structure. In order to illustrate the modeling of composition plans, let us consider that a particular realization of the ImageAssist component type of Figure 24 has two different component types to provide all of its functionalities. Such a composition plan includes a composite structure as presented in Figure 27.

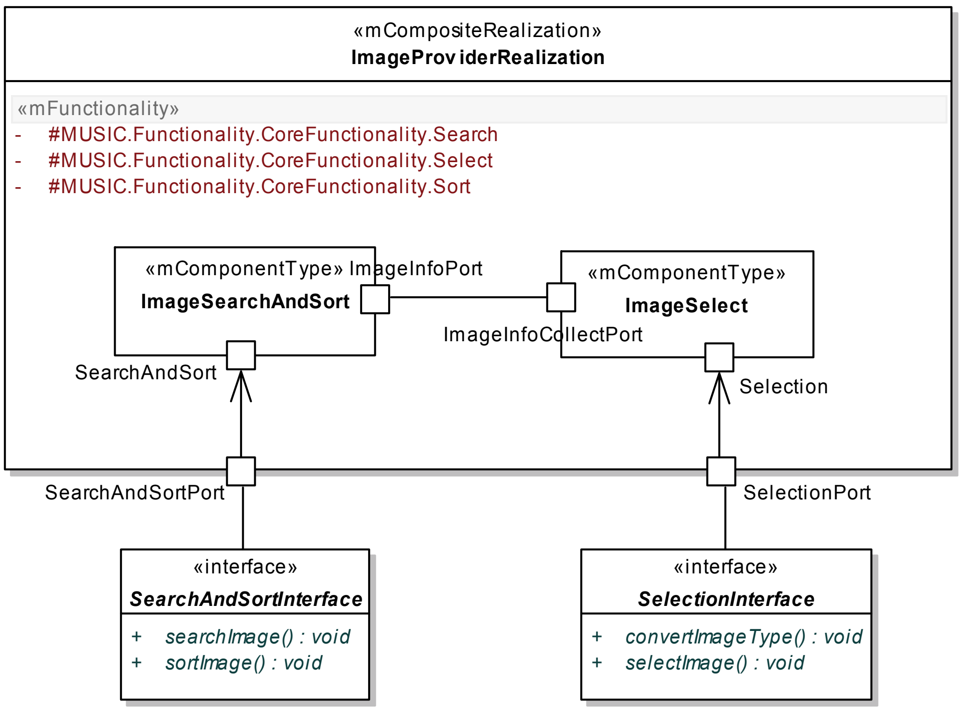


Figure 27: Modeling of a composite structure

In this figure, note that this realization provides one extra functionality ‘convertImage’ which is not required for the realization of the ImageAssist component type. However, as long as this realization can satisfy all the needed functionalities of the ImageAssist component, it may be used to realize that. Also, the type name written as ‘ImageProviderRealization’ does not need to match.

For this particular composition, the ImageSelect component (type) needs to communicate with the ImageSearchAndSort component (type), because the information about the sorted image might be retrieved from it. This is modeled by the internal connector. However, for some compositions this might not be necessary, especially when the component types in the composition are independent of each other. In addition to the composite structure, the developer of this particular plan has to provide also the definition of all the component types that he has used in the structure. But he does not necessarily provide the components or model further realization plans of those component types used in the composition. For example, an atomic realization plan for the ImageSelect component type along with the component may be provided by another independent developer, or it may be obtained from external services.

In the case of a realization of the application, part functionalities may be independent of each other; for example, the navigation functionality (see the scenario of section 1.2) is independent of searching and sorting images. In that case, the realization of the application, integrating components and services from different developers and third party service providers, becomes quite easy; especially, the matching of internally interacting interfaces can be avoided.

Modeling of plan: distribution

In order to be able to specify the adaptation capabilities with regard to the distribution aspect, deployment models are specified as separate deployment packages contained within the package of the composite realization. They include a deployment diagram containing the node type and the component types deployed on particular node types. By allowing the specification of more than one deployment models for a realization package, the modeling effort is reduced significantly: for a certain composition, the different possible distributions of the involved component types can be included in a single realization package.

In connection with the composition of Figure 27, we can think about two different deployment possibilities: 1) both component types are deployed on devices not belonging to Thomas, 2) the ImageSelect component might be light weight – consuming tolerable amount of resources – and may be deployed on Thomas's device. Such a case can be modeled as shown in Figure 28.

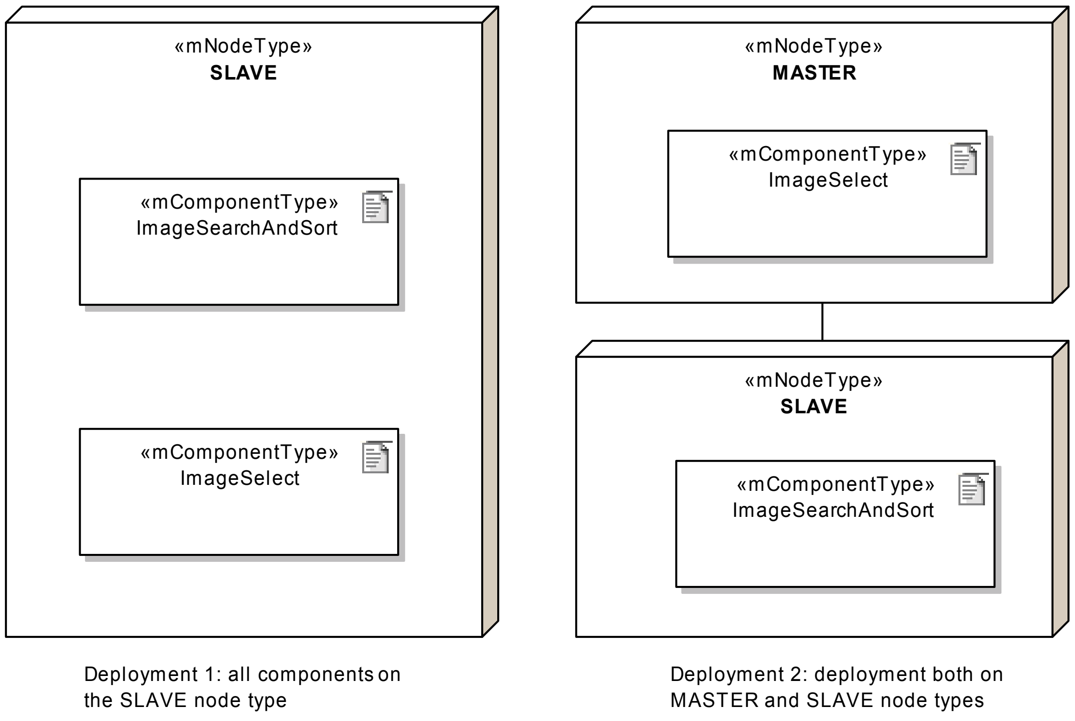


Figure 28: Deployment of component types on node types

Different deployments will most likely result in different utilities; but in the case of the unanticipated adaptation it might not be possible to always anticipate such deployment possibilities during the design of the plan. Different providers may offer the realization (component) of a particular component type. In that case, the actual utility may be dictated by the definition of the utility functions by the developer of that particular realization.

Modeling of plan: architectural constraints

The modeling of architectural constraints is unchanged compared to what is presented in the MUSIC deliverable D6.3 [63]. The scenario presented in this work does not use architectural constraints and the detail of the modeling technique is skipped in this document.

Modeling of plan: properties and utilities

This sub-task is dedicated to the identification of the required and provided property types of the application variants and alternative realizations of component types that are to be considered in the adaptation reasoning process. As the corresponding properties are compared to properties of the execution environment, these property types have also to be identified and to be linked to the MUSIC Domain Model. Afterwards, the properties for atomic and also for composite realizations have to be derived by defining constant properties or property evaluators/property predictors. The utility functions are also modeled within this subtask.

In the analysis phase an initial list of resource and context dependencies has been identified that has guided the application developer when defining realization plans of the application and component types. Thus, the initial list of dependencies gives a good hint at the property types to be considered in the adaptation reasoning process. In fact, all these context and resource dependencies should be reflected through the corresponding set of property types. So, if a context dependency tells that the alternative realizations differ in their required network bandwidth, for example, then the provided network bandwidth of the execution context and the network bandwidth required by the realization should be included as relevant property types in the adaptation reasoning process.

After the set of property types has been established, the atomic and composite realization packages have to be updated with the property specifications. This means to associate constant properties and to define property predictors/evaluators[17] that derive properties considering context information or parameter settings. Of course all the properties of the execution environment that are used in utility functions and in property predictors/evaluators have to be associated to concepts in the U-MUSIC Domain Model (section 7.2.2) and have to be retrieved from the Context Middleware. This may require updating the U-MUSIC Domain Model and developing additional context providers that are able to sense and deliver the requested context information.

The specification of properties and utility functions presented in this work differ to some extent from that presented in the MUSIC project. Properties and utility functions are dependent on the adaptation reasoning technique. In this work we present a new adaptation reasoning approach (see section 5.3), which does not require deriving properties for a composite from its constituent components. In this approach, we define

[17] Property predictors and property evaluators are synonymously used in this document.

utility functions for each realization plan rather than for the application as a whole. Therefore, for a composition plan, we need to define only those properties that influence the communications among different components in the composition, distribution of components etc. and not related to individual components. Clearly, the definition of utility functions for a composition also differs, where in the simplest case we need to just assign weight (importance) of a particular component (type) in the composition. These considerations are likely to reduce the usage of property evaluators and thus contributing to easier modeling.

The developer has to concentrate on modeling properties, property evaluators and utility functions for each of the realization plans. Properties may have a constant value or may be a function of other properties. An example of modeling constant properties for the atomic plan of the TextBasedUI component of Figure 26 is presented in Figure 29.

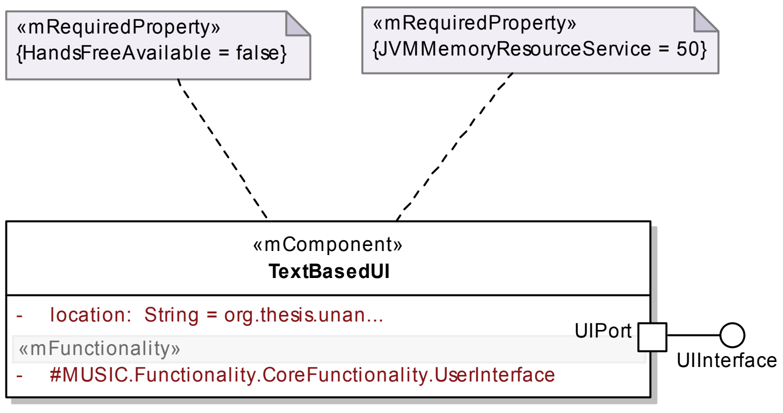

«mRequiredProperty»
{HandsFreeAvailable = false}
«mRequiredProperty»
{JVMMemoryResourceService = 50}
«mComponent»
TextBasedUI
- location: String = org.thesis.unan...
«mFunctionality»
- #MUSIC.Functionality.CoreFunctionality.UserInterface
UIPort
UIInterface

Figure 29: Modeling of constant properties

The specification states that the TextBasedUI component is supposed to be well-suited when there is no hands free user interface available; also it consumes 50 units of memory resource.

Compared to MUSIC, the usage of property predictors is greatly reduced, when we follow the adaptation reasoning approach, as presented in section 5.3. In our approach, we define utility function for individual plans and therefore, the properties of a composition do not need to be derived from its constituent components. Therefore, property predictors in this work are mostly used, when the expected value of a particular property type is not constant, rather it can be calculated as a function of values of other property types. An example of a property evaluator for a realization plan of the ImageSelect component type of Figure 27 is presented in Figure 30. (Please note that the ImageSelect component of Figure 30 realizes the ImageSelect component type of Figure 27.)

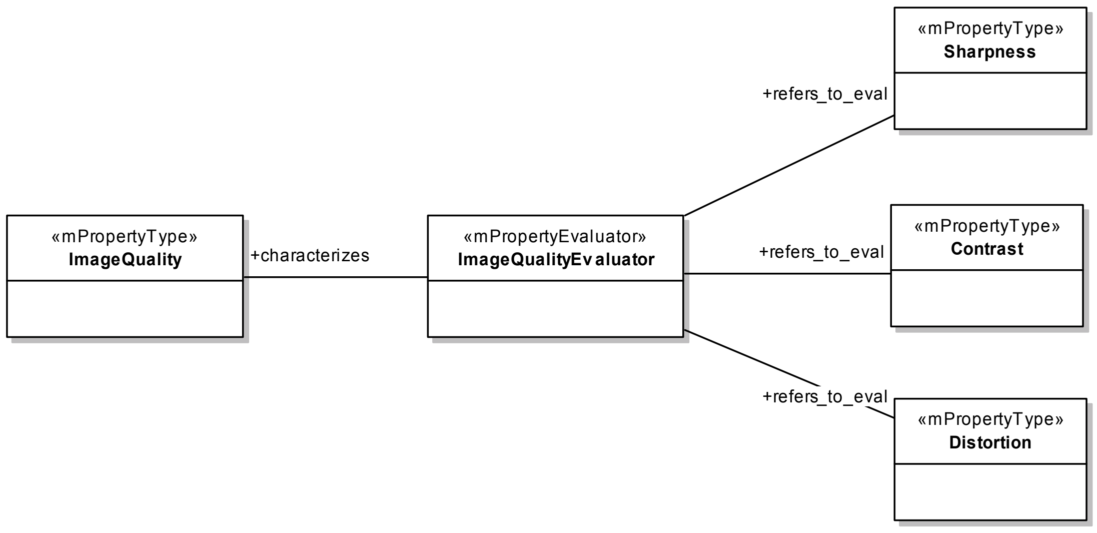


Figure 30: Modeling of property evaluators

The ImageQualityEvaluator property evaluator calculates the ImageQuality, identified by the 'characterizes' role in the model, using the values of image sharpness, contrast and distortion. The role 'refers_to_eval' at the connection end of a property type indicates that the property type refers to the particular component. This differs from the role 'refers_to_context', which indicates that the value of the concerned property type should be retrieved from the context information. The model only helps generating a skeleton for the property evaluator, while the actual calculation is added to the generated source code manually by the developer. However, pseudo-code may be added as notes, which appear in the generated source code as comments so that the developers may have an idea about the calculation to make.

The specification of utility functions for an atomic realization plan is simple. One example for the TextBasedUI component is presented in Figure 31.

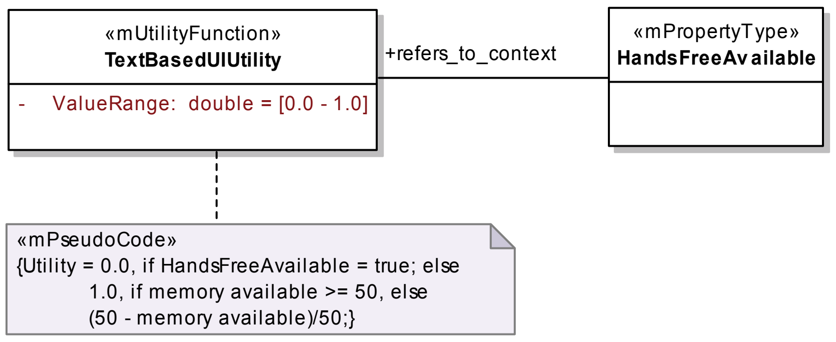

«mUtilityFunction»
TextBasedUIUtility
- ValueRange: double = [0.0 - 1.0]
+refers_to_context
«mPropertyType»
HandsFreeAvailable
«mPseudoCode»
{Utility = 0.0, if HandsFreeAvailable = true; else
1.0, if memory available >= 50, else
(50 - memory available)/50;}

Figure 31: Specification of the utility function for atomic realization plans

The specification of utility functions for composition plans depend on the kind of utility functions. In the simplest case, when the utility of the composition can be derived combining the weighted utilities of individual components, then just assigning the weights is enough; the transformation tool can automatically generate the appropriate source code. When the communication properties also influence the utility, then also in the model, assigning the weights is enough, but the generated source code has to be checked and filled out manually, if needed. The sum of the weights should be 1.0 in order to ensure that the calculated utility is normalized within the range of 0.0 and 1.0.

An example of the specification of the utility function for composition plans is presented in Figure 32.

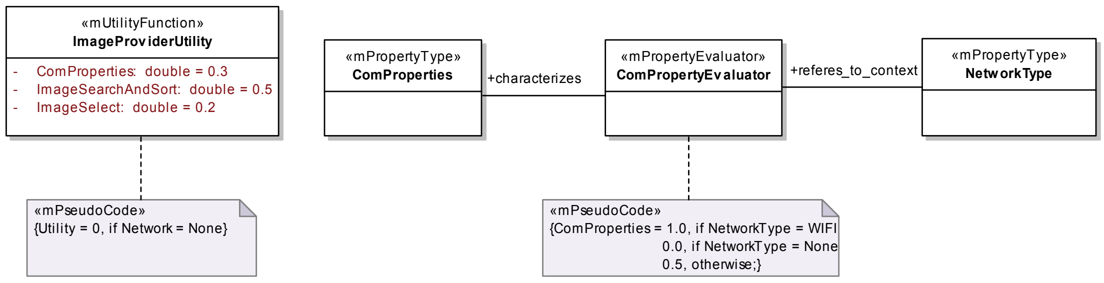


Figure 32: Modeling of utility function for composition plan

It defines the utility function for the composition plan of Figure 27. The utility depends on the part utilities of the ImageSearchAndSort and the ImageSelect component types. The respective importance/weights are 0.5 and 0.2. Moreover, the utility depends on the availability of the network. If there is no network, then this realization becomes useless and therefore, the utility is explicitly set to 0.0, as suggested by the pseudo-code. However, if the network is available, then the contribution to the overall utility of the composition is equal to 0.3 times the value calculated using the ComPropertyEvaluator property evaluator.

The modeling approach is not limited to specifying utilities as simple weights of the part utilities, rather any form of utility functions is supported, given, it is ensured that the utility value is normalized between 0.0 and 1.0. However, in such cases, only the skeleton of the utility function will be automatically generated and therefore, it needs to be filled out manually afterwards. The developer has to include proper guidelines in the model using pseudo-code.

MUSIC also provides a model driven development approach for developing context plug-ins [67], which we adopt from MUSIC without changes and therefore, not included in this document.

7.2.4 Model Transformation

A model transformation can be viewed as a transformation between two (or more) model spaces defined by their respective meta-models. Thus, transforming a source model to a target model is achieved by a transformation specification, which defines how a meta-model concept of the source model should appear in the target model. The transformation specification itself conforms to a meta-model as well; the latter defines transformation specification constructs. The input to the transformation tool is the 'Source', which in the MDA context is typically a Platform Independent Model (PIM). An MDA mapping typically provides specifications for transformation of a PIM into a PSM (Platform Specific Model) for a particular platform. The mapping is specified using some language to describe a transformation of one model to another. The description may be in natural language, an algorithm in an action language, or in a model mapping language. Code generation is a special case of model transformation where the output model is specified by means of an (executable programming) language. Tools are provided that generate executable code from the models via the PIM-PSM chain.

In the model driven development approach followed in this work, a platform-independent model of the application's adaptation capabilities is created. This model is transformed by appropriate tools to platform-specific source code publishing the artifacts (types, plans etc.) of application's variability model (section 7.2.3) to the middleware. We also generate source code for component skeletons; however, it does not include the functional aspects of the components and therefore, such component skeletons must be filled out manually.

Transformation of the variability model

In order to generate source code publishing the adaptation capabilities of an application to the middleware, the artifacts of the application variability model serves as input for the transformation tool. The transformation task relies upon the MOFScript language [68], which is a model-to-text transformation tool developed in the MODELWARE project [69]. The MOFScript language [68] [70] facilitates the generation of text (program code and XML, for instance) from MOF-based models, and it is related to the OMG standardization effort on MOF 2.0 Model to text [72]. MOFScript aims to be aligned with the principles of the QVT [71] and provides flexible mechanisms for generating text output. It is provided as an Eclipse plug-in. By using MOFScript language, a set of transformation scripts is developed for generating code that publishes adaptation capabilities to the middleware.

In order to use these scripts, the application developer has to make sure that the model he created is specified in the format of the Eclipse UML2 project, since MOFScript is based on Eclipse. For instance, a developer using Enterprise Architect [73] as a UML modeling tool may export the models he designed in the previous task to the corresponding XMI representations, by using an XSLT stylesheet (developed within MUSIC). Then, a developer may import this model on a proper Eclipse project and use the U-MUSIC transformation script on MOFScript for code generation. Another script is used to generate component skeletons.

7.2.5 Deployment

The variability model is the actual input to the transformation tool. Besides the definition of the component types and their alternative realizations, it also includes the definition of utility functions and property evaluators. Property evaluators and utility functions are not modeled in detail. Therefore, the model transformation can only result in skeletons for these functions. The actual body of the function has to be provided by the application developer and has to be hard-coded in the generated source code file. The missing gaps in the generated source code are marked with 'TODO' and the pseudo code fragments that are associated to the evaluators during the modeling task appear as comments to help the developers filling out the gaps. Afterwards, the generated classes have to be packed together with context plug-ins and components to derive OSGi bundles that are directly deployable on the middleware running on the target device.

Source code completion with property evaluators and utility functions

Utility functions and property evaluators are used to support the adaptation reasoning process. Theoretically, these functions are used to map application variants considering the current context situation and the required and provided properties of the application variant to a score (utility value). The objective is then to detect and select the variant which maximizes that score for a particular context. Property evaluators are used to derive the properties, which depend on other properties and/or parameter settings, in stead of having fixed values. In MUSIC, a single utility function is used to evaluate the fitness of a particular application variant, while in this thesis, each realization of a particular component type is evaluated separately and the utility of a composition is derived from the part utilities of its constituent components.

As already mentioned, the source code generated from the application variability model only contains nearly empty classes for the utility functions and the property evaluators. The corresponding classes can be identified by their names and the IPropertyEvaluator interface, which is used by the middleware components to evaluate properties and utility functions.

The application developer is required to implement the evaluate() method of the generated classes. This method uses the actual context information with the implementation-specific characteristics of the corresponding realization to compute the utility or the property.

The points of interest within the generated source code that require explicit attention are marked as 'TODO's. Application of the adaptation reasoning process adopted in this work is supposed to make the specification of utility function quite easy, although the number of utility function increases. For each atomic plan, they are defined using the QoS properties, while for composition plans, most of the utility functions, employing a linear combination of part utilities of the constituent components can be automatically generated. However, complex utility functions require manually filling out their source code. In our approach, the number of property evaluators is expected to reduce drastically, because the properties of a composition are no longer needed to be evaluated from the properties of the constituent components.

In the model, we do not suggest to include source code that can be directly integrated within the generated code. However, the developers may provide hints using UML Notes how the calculation of the utility function or property evaluator would look like. Most often, we could use some pseudo-code, which has to be manually translated into source code of the target programming language.

Sometimes it may be needed that some errors in the model are discovered, even after the generation and possibly filling it out with some code-fragments. That would require a new transformation after updating the model. In normal case, it would replace all the contents of the older file, eventually removing all the handmade updates. Fortunately, MOFScript provides the option to protect certain source code, when a file is replaced by a newer one, generated from a new transformation. The transformation script takes care of preserving manually added code (within certain blocks marked as //#Blockstart and //#Blockend) and therefore, the developer does not need to worry about it.

Packaging and deployment

When all the TODOs have their code completed, the application developer proceeds to develop or reuse any required context sensors. In addition, the developers can use any existing components that are required. For instance, if the application requires a context type to indicate the user's state (i.e., sleeping, walking, driving, in a meeting, etc.) then the developers have the option to either develop the necessary context plug-ins themselves, or locate an appropriate, existing plug-in (e.g. by browsing open-source repositories) and reuse it. The MUSIC project defines a Model-driven Development approach for context plug-ins as well [64] and we can use that approach also in our work.

After the necessary components and context sensors are available, the developer proceeds to create the OSGi [75] bundle that will be deployed on the middleware. OSGi bundles are used to distribute software to OSGi-compliant devices. These bundles are tightly-coupled, dynamically loadable collections of classes, JARs, and configuration files that explicitly declare their external dependencies (if any). An OSGi bundle may contain the following information:

- The class files and any other data that are used by the bundle to provide the services that are offered by the bundle.
- A file that describes the contents of the bundle which also includes parameter information to install and activate the bundle.
- A list of dependencies that the bundle requires to run. These dependencies are resolved before starting a bundle.
- A special class which is used to start and stop the services provided by the bundle and to perform any housekeeping required for starting or stopping the bundle.
- Optional documentation for the bundle or any of the subdirectories that is included in the bundle.

After the bundle is formed, the developer has the options to use either a precompiled version of the U-MUSIC middleware or to use the source of the middleware and to compile the U-MUSIC middleware by himself. The middleware is built using Maven [76], which allows the addition of URLs which can be used to download and build the latest modules or libraries required by the middleware.

At this point, the application developer has the bundles for the application he/she is developing as well as the U-MUSIC middleware which will be used to deploy the application on. The last thing is the deployment of the application and the launching of its services. This can be achieved in two ways using the middleware:

- The application developer can use the GUI to start and stop the application explicitly and,
- The bundle can be deployed by two ways:
    1. The mechanism provided by the OSGi framework, and/or

2. The middleware GUI, which can be used to install/uninstall bundles, when the middleware is already running.

For more details on how to compile the middleware (both MUSIC and U-MUSIC), how to create appropriate OSGi bundles and how to start the services we refer to the document MUSIC Development Environment [87].

7.2.6 Testing and Validation

Testing of self-adaptive systems aims at validating the reasoning of a given system to ensure that correct adaptations take place when the execution context evolves. Thus, the testing method requires controlling the execution environment in order to describe and execute the scenarios of validation. This control covers both the simulation of context situations to support the context evolution, and the simulation of client profiles that are involved in the system. The control of these input parameters allows the isolation of the self-adaptive system execution context.

The scenarios of validation should define which adaptation actions should be taken by the reasoning engine for each identified context situation. The description of a scenario basically includes the description of (i) an initial context situation, (ii) a context evolution and (iii) the expected adaptation. The initial context situation is identified by a set of context data. Then, the context evolution can be described as a new context situation or as the change of a set of context data. Finally, the expected adaptation can be expressed as the resulting state of the self-adapting system after the execution of adaptation actions.

To report the result of a given adaptation, it is necessary to observe the reasoning process for being able to analyze both the reactions and the decisions of the adaptation engine when the context is evolving. The analysis of this observation allows the detection of conflicting adaptation policies and provides support to the developer to refine the adaptation policies and improves the accuracy of the reasoning.

In order to tune the adaptive behavior of applications the utilities of different application variants have to be analyzed and the weights and properties contributing to the utility have to be adjusted. For this purpose, we propose the following steps:

- Select important property predictors that should be tuned
- Develop testing scenarios for the tuning of the selected property predictors
- Perform extensive simulations that allow the collection of measurements from which the needed adjustments of properties and property predictors can be derived

Finally, the test of self-adapting systems requires also an evaluation of the cost of the adaptation process. This means that the performance of the system should be monitored in terms of memory footprint, runtime latency, energy consumption, and processor occupation. This ensures that the self-adapting system respects the specification of the platform on which it should be deployed. We should take into account the overhead created with this monitoring.

The MUSIC Studio provides a set of tools to aid the testing and validation of adaptation behavior and performances. However, such support is still at a primary state in terms of adaptation testing and validation. In this thesis we have tested the middleware for the initial support of unanticipated adaptation and the performance of the adaptation reasoning process (section 5.3).

7.3 Tool Support

The methodology for the development of unanticipated adaptive applications, as presented in section 7.2, requires tool support at various steps. In fact, the success of the development approach is highly dependent on using tools to speed up the development process as well as for error-free development. Different tools are required at different development steps, e.g., to create application adaptation model, generate source code, test and validate adaptive behavior etc. The MUSIC Studio [74] is a suite of those tools integrated together to help application developers in creating applications based on the MUSIC middleware. This suite contains a mixture of selected pre-existing open source (preferable) tools and custom developed tools for the MUSIC project. A top level view of the MUSIC Studio is presented in Figure 33.

For this thesis, we have used most of the tools as they are provided by the MUSIC project; however, because of the changes in the modeling methodology, the UML2JavaTransformation tool is updated. After a brief overview of the Studio, each of the tools will be introduced, while the main updates of the UML2JavaTranformation tool will be highlighted. The details of each tool are out of the scope of this document and interested readers are suggested to follow MUSIC WP7 deliverables [74].

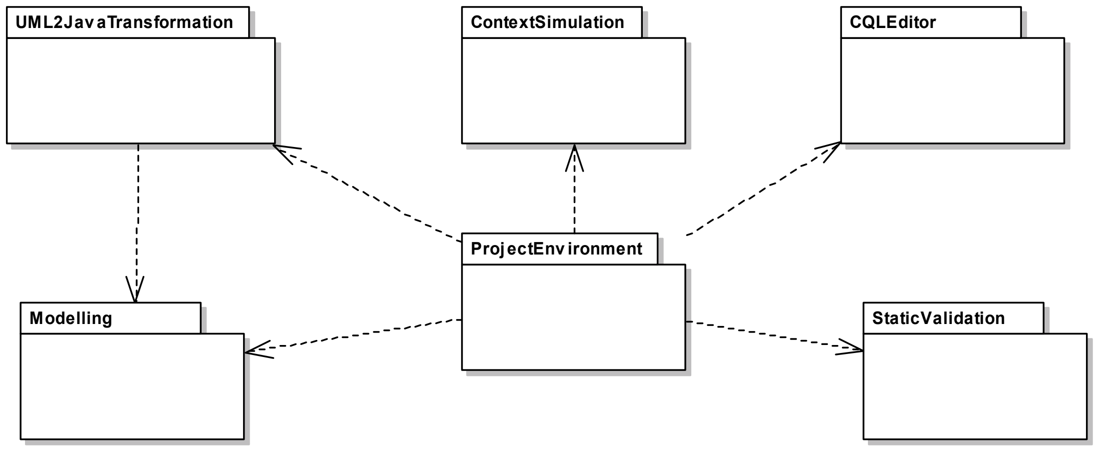


Figure 33: Top level view of the MUSIC Studio

The Project Environment is not a specific tool; rather it assists the developer in setting up a project for developing applications and components, and can contain e.g., templates and wizards which set up all required files for a project involving the full tool chain of the MUSIC Studio. Therefore, the Project Environment has a dependency to each of the tools used in the Studio.

The Modeling tool is used for creating UML models of the U-MUSIC application variability, using a UML profile. For our work, the same UML profile from the MUSIC project is used, although the modeling methodology is different.

The CQL Editor provides tool support for the Context Query Language. CQL is an XML-based language, whose syntax is described by an XML Schema Definition (XSD). This means that most existing XML editors, which understand XSDs, can provide basic CQL support.

The UML2JavaTransformation tool transforms models created using the Modeling tool to a representation useable by the U-MUSIC middleware. Such tools are needed for transforming the variability model of the application as well as for creating component skeletons.

The Static Validation tool validates (i.e. checks) application models in order to detect errors and omissions. The main goal is to ensure that developers have filled in what is needed in order to achieve a working adaptation model. It helps catching some design errors that are introduced in the source code and would otherwise be manifested during the runtime execution, such as those related to the property evaluation (including utility functions). Examples of validation are: whether the related context values or property values are defined, whether the context value types and property value types are correct, whether the MANIFEST.MF file is correctly defined etc.

The Context Simulation tool is a part of a prototype test and simulation environment, to allow developers to observe and analyze the effects of context changes and adaptations. Also, we need to provide visual information on the state of the middleware and applications and their actions. Moreover, it may be needed to enable early testing of the values of property predictor functions. Such features are provided by this tool as well.

An Eclipse-based implementation of MUSIC tools is preferred, when feasible, for easy integration within the MUSIC Studio. From that perspective, MUSIC tools are managed through an update site in order to simplify installation of the MUSIC studio within Eclipse, by providing the middleware and tools as Eclipse Features through the Eclipse update mechanisms.

In the following sub-section, we provide a process model for each of the tools in order to briefly explain the functionality expected to be provided by them.

7.3.1 Modeling Tool

The Modeling tool is used to create the specification of application types, component types and plans in the UML. A process model of the tool is presented in Figure 34.

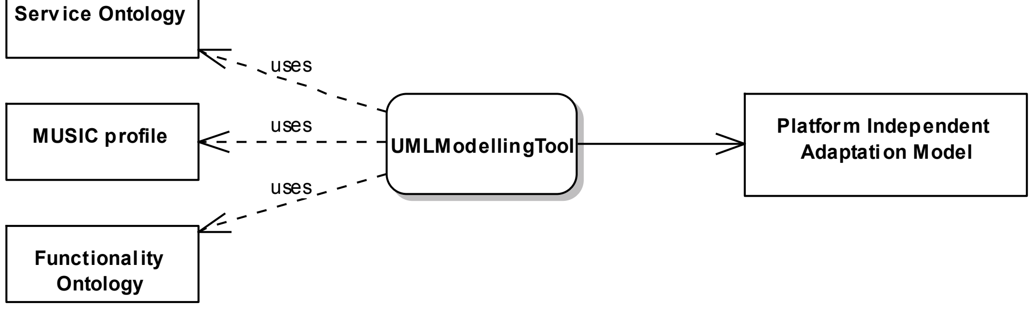


Figure 34: Process model for the modeling tool

The Modeling tool uses the MUSIC UML profile (which is sufficient for this work, too), the MUSIC ontology for Context and Services as provided by MUSIC, with the possibility of extending it by the developer. In addition to this, the developer of a U-MUSIC application needs to provide a Functionality Ontology, which helps addressing functionalities that his applications and/or components related to.

Based on the steps discussed in the methodology, a Platform Independent Adaptation Model in UML is created. The Ontologies are described using OWL (Web Ontology Language) and for that purpose we have used the Protégé Ontology Editor [77], which is an open source tool. The adaptation model in UML can be exported to the xmi [79] format to facilitate the transformations later. Most of the UML modeling tools also provide this feature.

The created model may be compliant either to the OMG UML2.1 or to the EMF (Eclipse Modeling Framework) UML2 meta-model, which is an EMF-based implementation of the OMG version of the UML 2.x meta-model. Many tools support modeling compliant to both meta-models, while most of the Eclipse-based modeling tools support modeling corresponding to the UML2 meta-model. In MUSIC, Enterprise Architect has been used as the primary modeling tool, mainly because of its user-friendliness and low cost and the fact that no fully-featured open source tool was available. However, currently we have found an open source tool in Papyrus [78] that can replace Enterprise Architect. Papyrus is preferable, because it is an open source tool and also being an Eclipse plug-in, it can be easily integrated within the MUSIC Studio. Therefore, models compliant to the EMF UML2 are a more straight-forward solution. The MUSIC project provides two formats of the MUSIC profile, one in OMG UML2.1 and the other in EMF UML2, while the first one can be imported in Enterprise Architect to use the notation and the second one can be either imported or used as an Eclipse plug-in to be used during the modeling by an Eclipse-based modeling tool.

7.3.2 CQL Editor

The MUSIC project has developed a new Context Query Language (CQL) [105] that addresses the application area of mobile and pervasive computing, where aspects like autonomy, distribution, mobility, heterogeneity etc. need to be supported by a query process asking for context information. The CQL is based on XML and strongly related to the underlying context model, which again uses the MUSIC Ontology. The CQL Editor aids specifying context queries. The process model of the CQL Editor is simple, as it is presented in Figure 35.

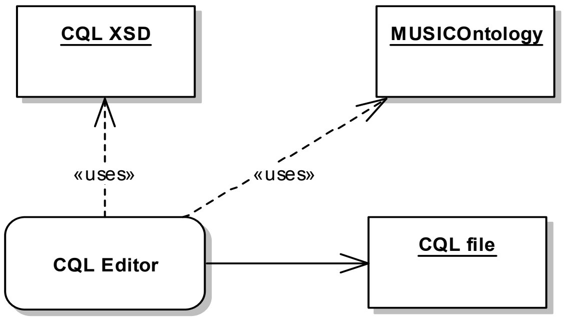


Figure 35: Process model of the CQL Editor

The CQL Editor is a regular Eclipse plug-in and therefore, it can be installed by copying its jar file to the Eclipse plugins folder. It uses the CQL XSD based on the Context Query Language provided by the project and the result is a CQL file with the extension of .cql.

The CQL Editor possesses a number of characteristics:

- A new file wizard automatically generates an example CQL file and links to the CQL XSD to provide code completion. The developer may take it as a basis and complete the file according to his need.
- The CQL XSD is contained in the plug-in, so it does not have to be installed or copied separately. The XSD is also registered in the Eclipse XML register.
- The .cql extension is registered so that CQL files are known by Eclipse and are opened in the XML editor by default.

7.3.3 UML2JavaTransformation Tool

The Platform Independent Model developed using the Modeling tool needs to be transformed into Platform-specific Model and/or source code. For our task, we have modeled the adaptation capability of the application in UML and use Java as the language of the target source code. Such functionalities are available in many UML Modeling tool as well; however, such transformation does not perfectly fit with our purpose and therefore, we need a separate UML2Java transformation tool. In Figure 36 the process model of the UML2JavaTransformation tool is presented.

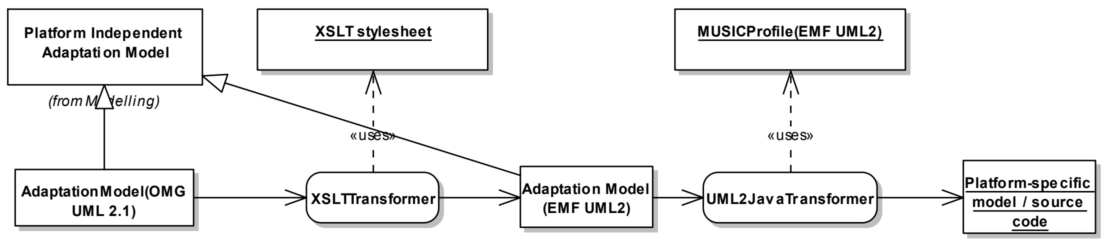

Platform Independent Adaptation Model
(from Modelling)
XSLT stylesheet
MUSICProfile(EMF UML2)
«uses»
«uses»
AdaptationModel(OMG UML 2.1)
XSLTTransformer
Adaptation Model (EMF UML2)
UML2JavaTransformer
Platform-specific model / source code

Figure 36: Process model for the UML2JavaTransformation tool

The tool selected for generating source code in MUSIC as well as in this work is MOFScript [68], which is an implementation of the MOFScript model to text transformation language. The tool is available as an Eclipse plug-in and supports parsing, checking, and execution of MOFScript scripts. However, the modeling tool Enterprise Architect (EA) does not support storing models in an Eclipse-compatible format, so it is necessary to transform the UML application models from EA to the Eclipse format. Consequently, the transformation chain becomes a two-step process, when the adaptation model is prepared in an OMG UML2.1 tool like EA:

1. An XSLT transformation translates XMI-output from EA into an EMF UML2 model.
2. MOFScript transformations generate java source code from this EMF UML2 model.

By using this two-step process, the important Java-generating transformations remain independent of EA. This means that, in case we decide to switch to another modeling tool, those can still be used without any problem. For example, if we use Papyrus, which provides the facility of modeling in EMF UML2, then the XSLT transformation step is not required.

There are two scripts in MOFScript to transform the UML model customized by the XSLT stylesheet to produce source code in Java. The first one is used to produce the specification of adaptation plans, application types and component types. It creates a file containing a class that implements the U-MUSIC IBundle interface (see section 6.1.1). The goal of the latter script is to generate as much as possible of the components' code. The logic itself has to be coded manually, but the script will generate skeleton classes. This is useful both for saving manual work as well as making sure the classes are consistent with the models – which represent the first class entities of the application.

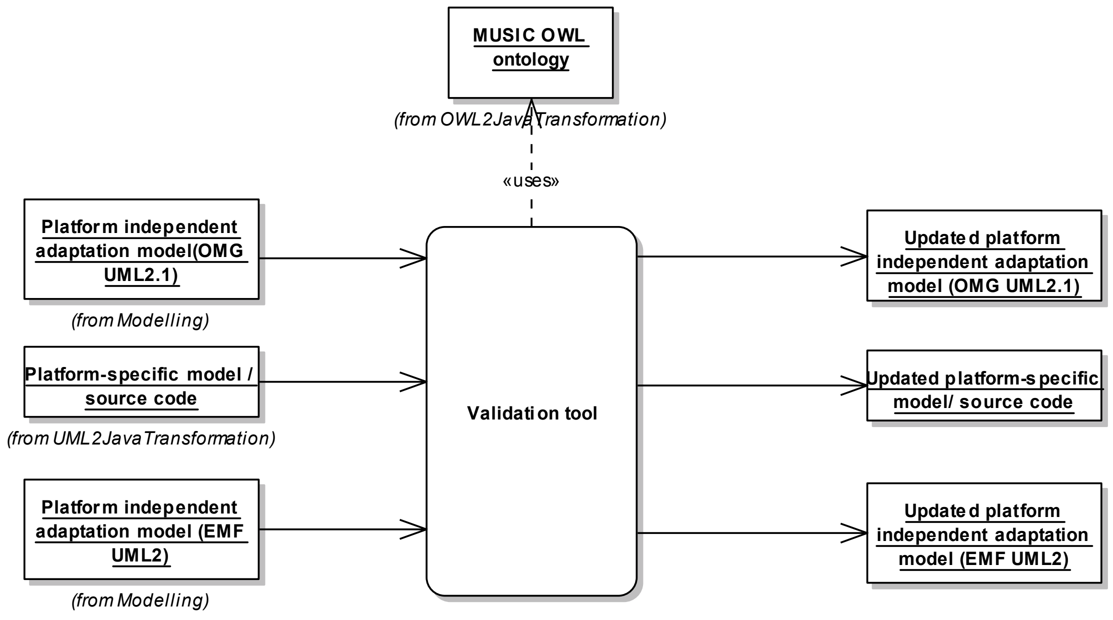
Extensions to MUSIC

Since the model in this work differs from that of MUSIC, the following updates are made to the MUSIC transformation script:

- In the MUSIC model, there are static references between a plan and a type through the realization dependencies. Therefore, in the script such plans are first collected as set of realization plans for that particular type. In this work, such references are removed and therefore, types and plans are collected from the model separately, while generating the source code.
- Unlike MUSIC, utility functions are specified for each plan. Therefore, the generation of utility function is updated.
- In general only the skeletons of such utility functions are generated and they need to be filled out manually. In this work, we maintain that. However, we ease the task for a special case, i.e., when for a composition plan, the utility function is a summation of part utilities of its constituent component (types in the model), then the complete utility function is generated automatically and it does not require developer intervention afterwards.

7.3.4 Static Validation Tool

The Static Validation tool uses the information in the MUSIC ontology to validate both the PIM and PSM against the unavailability of required context and property information. The tool indicates the errors and may suggest a possible reason for such error; but it does not provide an automatic correction of errors. The process model for the tool is presented in Figure 37.



MUSIC OWL ontology
(from OWL2JavaTransformation)
«uses»
Platform independent adaptation model(OMG UML2.1)
(from Modelling)
Platform-specific model / source code
(from UML2JavaTransformation)
Platform independent adaptation model (EMF UML2)
(from Modelling)
Validation tool
Updated platform independent adaptation model (OMG UML2.1)
Updated platform-specific model/ source code
Updated platform independent adaptation model (EMF UML2)

Figure 37: Process model for the static validation tool

The tool is designed as an Eclipse plug-in and will be installed as a separate plug-in project in the same workspace as the application projects that need to be validated. The current tool is based on MUSIC middleware v0.1.1. In addition to a validation tool bundle, the plug-in also includes a modified adaptation bundle of the middleware.

At the start up, the validation tool displays all the available applications and allows the user to select the applications for validation. For the selected applications, validation output is presented. Currently, the validation output is mainly the log of the execution of the validation tool, showing information such as the progress of validation, the model states (e.g. the number of variants), runtime context element types and values as well as validation errors such as undefined property values and undefined context values. The use of the current implementation is very limited; however, work on enhancing the tool is in progress in the MUSIC project.

7.3.5 Context Simulation tool

The Context Simulation tool generates context changes that are fed to the middleware and trigger the adaptation decision-making process. The middleware passes information about the internal working of the adaptation decision-making process back to the tool and this evaluation result is presented to the tester. The tester evaluates whether or not the correct adaptation decision has been made. The evaluation result may also include the information on middleware states and actions based on the context changes. Moreover, it may contain runtime values of properties and utilities, given particular utility functions and property predictors are provided as inputs. The process model of the tool is presented in Figure 38.

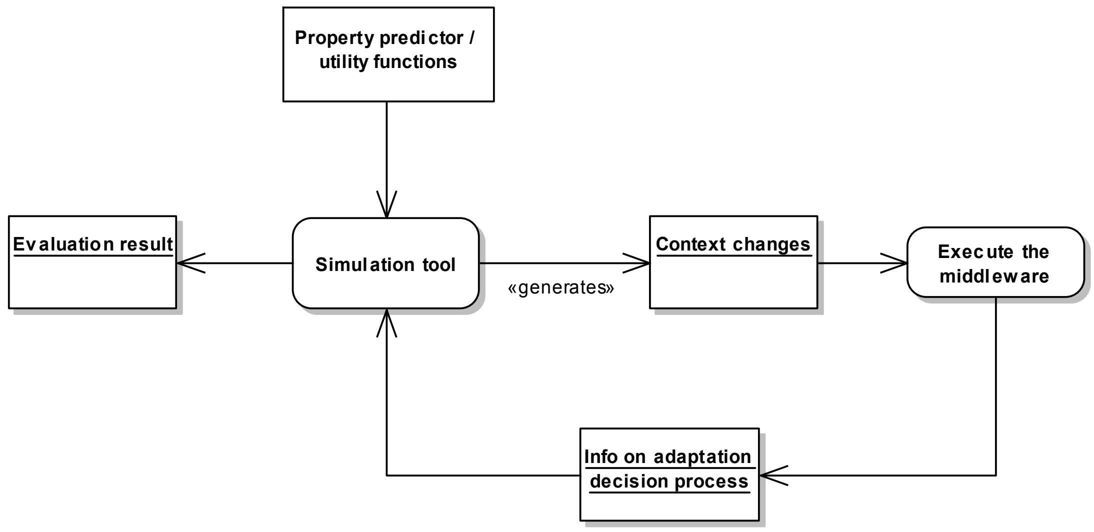


Figure 38: Process model for the context simulation tool

The tool collects sufficient information so that, in the event of an incorrect decision being made, an investigation can determine the source of any problems. The problem may be due to incorrectly defined platform independent UML models, errors in the process that transforms the UML models into a platform specific computerized model (Java code), errors in the way the middleware handles the computerized model, errors in the utility function and/or errors in context handling.

The tool is most commonly used in a controlled test bed environment. Because the tool is only interested in the internal decision making processes of the adaptation and not application functionality, it can be used for testing the application's adaptation responses before application functionality development is complete.

The old GUI from the MADAM project provides a chance of simulating simple context. However, that GUI is replaced in MUSIC and currently Context Simulation tool is under development. The context simulation tool will be a standalone tool and will not be packaged as part of the MUSIC Studio. This is because the MUSIC Studio tools are for use at design-time, not runtime.

For this thesis, we have not made any update to any of the tools, except the transformation tool. Therefore, we have not investigated the possible updates needed in other tools of the MUSIC Studio. For example, the change in the adaptation reasoning approach modifies the way utility functions and property evaluators are used. Most likely, it would require an update to the Context Simulation Tool; but we have not investigated such issues. Moreover, most other tools, except the modeling and the transformation tools, in the MUSIC Studio are still in their infancy and therefore, the tests performed in this thesis (chapter 8) mainly use the Modeling tool and the UML2JavaTransformation tool, while the MUSIC GUI is used to observe adaptation and take readings of adaptation reasoning and configuration time.

Part III Evaluations and Conclusions

8 Test Applications

Probably the most attractive way of testing the unanticipated adaptation as presented in this work would be implementing the UnanticipatedTravelAssistant application and demonstrating the scenario of section 1.2. However, in this work we are more concerned with the adaptation technique than the implementation of the functionalities of the components. Therefore, we avoid the development of such an application with a real world demonstration requiring too much effort; rather we demonstrate the proof of concepts described in this thesis in an easier way. For interested readers, we would like to refer to MUSIC WP2x deliverables (http://www.ist-music.eu/MUSIC/results/music-deliverables), which describes a number of trial applications to developed in MUSIC to demonstrate the adaptation capability.

In this work, we mainly focus on the following two aspects that are not covered by MUSIC:

1) The unanticipated adaptation by developing components independently

2) The adaptation reasoning technique for a huge number of application variants.

For the first aspect, we have developed three independent bundles, the first one containing an application type and a plan, the second one containing two plans and the third one containing a single plan. As a support of the unanticipated adaptation, it is checked, if these plans providing the functionalities of the application type can be used to realize the application.

For the second aspect, we have developed two arbitrarily large application variability model; one producing 2,004,697 (~2 million) application variants and the other being able to create 15,595,417 (~15.6 million) variants. The performance of the adaptation reasoning approach is checked. The components developed for these applications do perform only the simple task of presenting some console output to indicate their presence in the application configuration.

8.1 Testing the Unanticipated Adaptation Behavior

In order to test the unanticipated adaptation behavior, we take tutorials 1 and 2 from the MUSIC project as the baseline. Both the tutorials are quite simple; the first one uses a single atomic realization plan for the application, while the second one uses two different plans. The orientation of the device screen is used as the context influencing the selection of a plan. For the first tutorial, only a single plan is used; but based on the orientation, the component provides console output, whether the orientation is landscape or portrait. For the second tutorial, the console output depends on the selection of the plan.

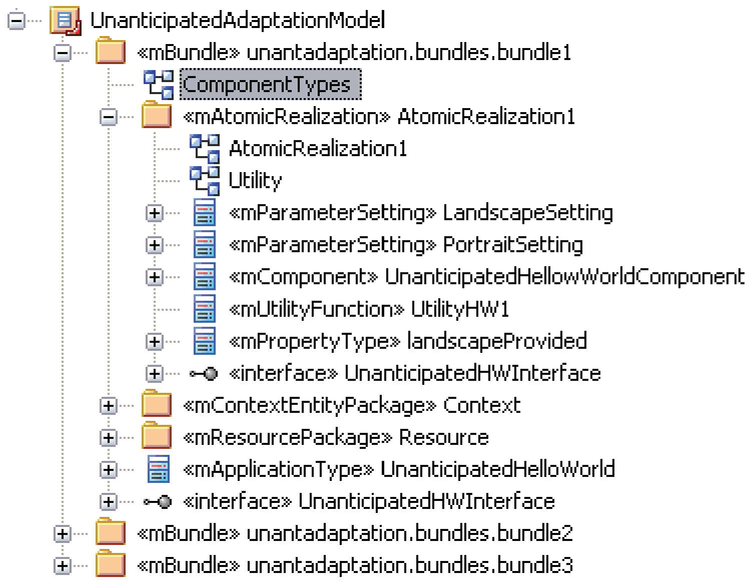
For this test, we have independently developed three bundles, the first one contains the definition of the application type and one plan (corresponding to tutorial 1), the second one contains only two plans (corresponding to tutorial 2) and the third one contains a single plan (slightly changed tutorial 1). However, unlike MUSIC, the correspondence between the application type and the plans are not established at design time; rather they are established, when a new bundle is deployed, while the application is running.

8.1.1 Bundle 1

The bundle is developed applying the U-MUSIC Model Driven Development approach as described in section 7.2. In the following, we present the adaptation model, transformation and its packaging as a bundle.

Model

The model follows a predefined structure, which is required to apply the transformation tool successfully. The model structure is flexible enough to accommodate modeling adaptation capabilities for application of any size, while there are many optional diagrams, which can be bypassed for simpler applications. The diagram structure for this model is presented in Figure 39.



```
UnanticipatedAdaptationModel
  «mBundle» unantadaptation.bundles.bundle1
    ComponentTypes
    «mAtomicRealization» AtomicRealization1
      AtomicRealization1
      Utility
      «mParameterSetting» LandscapeSetting
      «mParameterSetting» PortraitSetting
      «mComponent» UnanticipatedHellowWorldComponent
      «mUtilityFunction» UtilityHW1
      «mPropertyType» landscapeProvided
      «interface» UnanticipatedHWInterface
    «mContextEntityPackage» Context
    «mResourcePackage» Resource
    «mApplicationType» UnanticipatedHelloWorld
    «interface» UnanticipatedHWInterface
  «mBundle» unantadaptation.bundles.bundle2
  «mBundle» unantadaptation.bundles.bundle3
```

Figure 39: Structure of packages and diagrams

The bundle is defined within a package stereotyped as 'mBundle'. It can contain any number of class diagrams to define application types and component types as well as any number of realization plan packages. Each bundle also contains a context entity package and a resource package to define the related context and resources. This figure also shows the other two bundles in the same modeling file; however, they can be developed in separate files. During the transformation, each individual bundle is transformed separately, anyway.

The model starts with defining the application type in the class diagram, named 'ComponentTypes'[18] as shown in Figure 40.

[18] Diagram names are arbitrary and do not have any influence on the transformation.

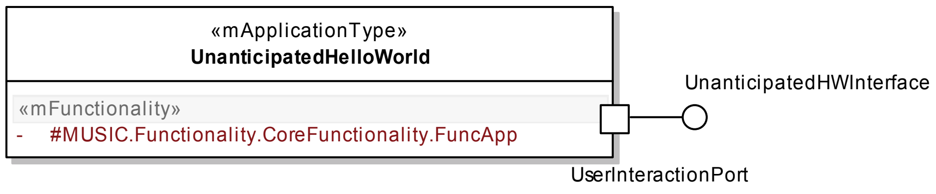


Figure 40: Specification of the UnanticipatedHelloWorld application type

The application defines a single functionality 'FuncApp' (it appears in the generated source code as *http://www.ist-music.eu/Ontology_v0_1.xml#Functionality.CoreFunctionality.FuncApp,* where, *#MUSIC* in the model is replaced by the reference to the ontology) so that any plan corresponding to a component or service providing that functionality would realize the application. The model of the atomic realization plan for the UnanticipatedHelloWorldComponent is presented in Figure 41.

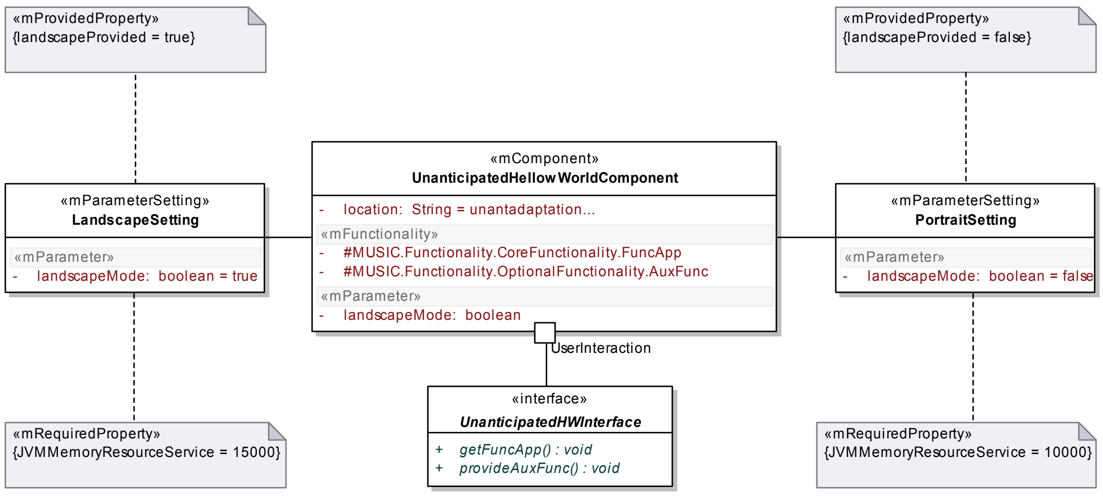

«mProvidedProperty»
{landscapeProvided = true}
«mProvidedProperty»
{landscapeProvided = false}
«mComponent»
UnanticipatedHellowWorldComponent
- location: String = unantadaptation...
«mFunctionality»
- #MUSIC.Functionality.CoreFunctionality.FuncApp
- #MUSIC.Functionality.OptionalFunctionality.AuxFunc
«mParameter»
- landscapeMode: boolean
«mParameterSetting»
LandscapeSetting
«mParameter»
- landscapeMode: boolean = true
«mParameterSetting»
PortraitSetting
«mParameter»
- landscapeMode: boolean = false
UserInteraction
«interface»
UnanticipatedHWInterface
+ getFuncApp() : void
+ provideAuxFunc() : void
«mRequiredProperty»
{JVMMemoryResourceService = 15000}
«mRequiredProperty»
{JVMMemoryResourceService = 10000}

Figure 41: Model of the UnanticipatedHelloWorldComponent plan

The component provides the 'FuncApp' functionality in addition to the extra functionality 'AuxFunc'. At runtime, it will be checked and match to create the application variability model. In general, the matching should be carried out using the information in the Ontology and in the case of imprecise matching the functionality string does not necessarily have to match 100%. However, for this initial implementation of the middleware, we only perform a string matching. The properties and resource requirements of this realization are parameterized. There are two different parameter settings and corresponding to each of them properties and resource requirements are modeled.

The specification of the utility function is simple as presented in Figure 42.

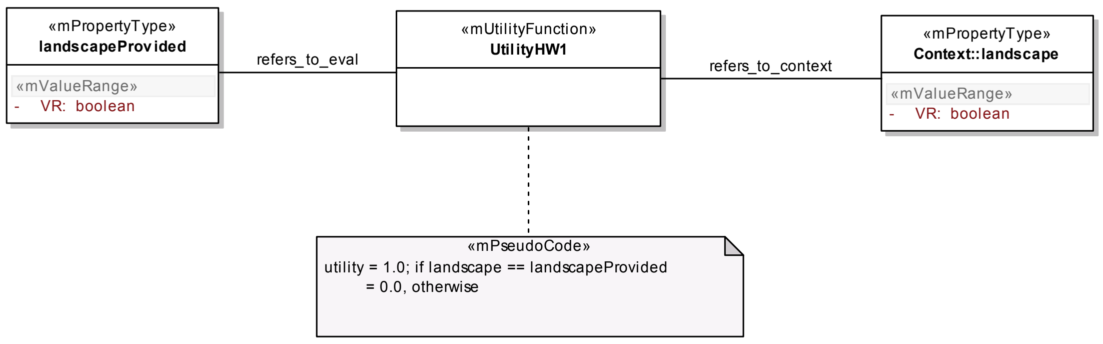


Figure 42: Utility function for the atomic realization plan of Figure 41

The utility depends on the context property type ‘landscape’ and the evaluator-specific property type ‘landscapeProvided’. The model helps auto-generating a skeleton of the utility function and the pseudo-code guides the developer when he has to fill out the generated source code manually.

The reference to the context is specified with the help of the context model as presented in Figure 43.

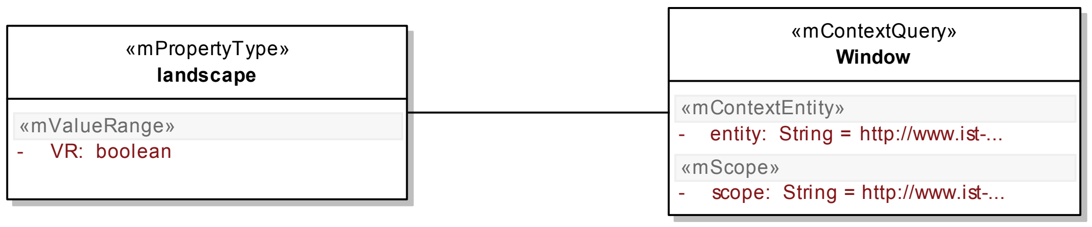


Figure 43: Modeling context properties and queries

In relation to the property type ‘landscape’ a context query defines the context entity in concern and its scope. References to context entities and scope depend on the MUSIC Ontology. For this example, ‘*http://www.ist-music.eu/Ontology_v0_1.xml#Concepts.Entities.Device.Screen*’ is the reference to entity and scope refers to ‘*http://www.ist-music.eu/Ontology_v0_1.xml#Concepts.Scopes.Screen.Landscape*’. Therefore, when the value of the ‘landscape’ property type is queried for, it automatically refers to the ‘Screen’ entity with the ‘Landscape’ scope in the MUSIC Ontology. Property types related to resources; e.g., JVMMemoryServiceResource is modeled similarly within the package stereotyped as «mResourcePackage».

Generated source code

Using the UML2JavaTransformation tool (section 7.3.3), the bundle is transformed to create corresponding Java source code. Based on the bundle name, a Class implementing the U-MUSIC IBundle interface is generated. The source code corresponding to the application type is presented in Figure 44.

```
public class Bundle1 implements IBundle {
    // Type Names
    private static MusicName UnanticipatedHelloWorld = MusicName.nameFromString("" +
            "/type/unantadaptation.bundles.bundle1/UnanticipatedHelloWorld");

                                        … …

    public ApplicationType[] getApplicationTypes(){
        ApplicationType[] application = new ApplicationType[] {
            new ApplicationType(UnanticipatedHelloWorld, new String[]{"http://www.ist-music.eu/
                Ontology_v0_1.xml#Functionality.CoreFunctionality.FuncApp"}, null) };
        return application;

                                        … …
}
```

Figure 44 Source code fragment corresponding to the model of the application type

The application type has a type name along with the functionalities and an array of properties. Such properties are set at runtime based on the status of the application; e.g., whether it is running or suspended or stopped. The generated source code corresponding to the atomic realization plan is presented in Figure 45.

```
// Names
private static String UnanticipatedHelloWorldComponent_Name =
        "UnanticipatedHelloWorldComponent";
//Create and install plans
private AtomicPlan[] getAtomicPlans(){
    AtomicPlan[] ATOMIC_PLANS = new AtomicPlan[1];
    String[] contextDep_0 = {
        new String ("http://www.ist-music.eu/Ontology_v0_1.xml#Concepts.Entities.Device.Screen;
            http://www.ist-music.eu/Ontology_v0_1.xml#Concepts.Scopes.Screen.Landscape")
    };
    ATOMIC_PLANS[0] = new AtomicPlan(HelloWorldComp_Name, new String[]{"http://www.ist-
            music.eu/Ontology_v0_1.xml#Functionality.CoreFunctionality.FuncApp", "http://www.ist-
            music.eu/Ontology_v0_1.xml#Functionality.CoreFunctionality.AuxFunc"}, null,
            "unantadaptation.bundles.UnanticipatedHelloWorldComponent", contextDep_0);
    {
        Map propertyMap = myCreateMap(
            new String[]{"landscapeProvided", IPropertyEvaluator.UTILITY_PROPERTY },
                new Object[]{
                    new ConstProperty(new Boolean (true)),
                    new Utility()});

        Map resourceMap = myCreateMap(
            new String[]{"JVMMemoryResourceService"},
                new Object[]{
                    new Integer(15000)});

        Feature[] features = {
        };
        {
            Map parameterSettingsMap = myCreateMap(
                new String[]{"landscapeMode"},
                    new Object[]{
                        new Boolean(true)});

            ATOMIC_PLANS[0].addPlanVariant(propertyMap, parameterSettingsMap, resourceMap,
                    features);
        }
    }

    {
        Map propertyMap = myCreateMap(
            new String[]{"landscapeProvided", IPropertyEvaluator.UTILITY_PROPERTY },
                new Object[]{
                    new ConstProperty(new Boolean (false)),
                    new Utility()});

        Map resourceMap = myCreateMap(
            new String[]{"JVMMemoryResourceService"},
                new Object[]{
                    new Integer(10000)});

        Feature[] features = {
        };
        {

            Map parameterSettingsMap = myCreateMap(
                new String[]{"landscapeMode"},
                    new Object[]{
                        new Boolean(false)});
                ATOMIC_PLANS[0].addPlanVariant(propertyMap, parameterSettingsMap, resourceMap,
                        features);
            }
        }
        return ATOMIC_PLANS;
}//getAtomicPlans()
```

Figure 45: Source code fragment corresponding to the atomic plan

The context dependency of the plan is retrieved from the properties, their context query and the relation to the Context Ontology as specified in the model. The transformation tool automatically organizes the combination of parameter settings, properties, and resource requirements into different sets to create variants of the plan (see section 6.1.1). Plan variants can be considered as plans providing the same set of functionalities, using the same component, while requiring to instantiate the component with a different set of properties, parameter values etc. The functionalities provided by the plan are also required to construct the plan. Utility functions are referred to as properties; however, they are identified as 'IPropertyEvaluator.UTILITY_PROPERTY'. The generated source code corresponding to the utility function is presented in Figure 46.

```
class UtilityHW1 implements IPropertyEvaluator{
    public Object evaluate(IContextValueAccess context, IPropertyEvaluatorContext evalContext)
    {
        double utility = 0.5;
        boolean landscapeProvided = ((Boolean) evalContext.evaluate("landscapeProvided",
                context)).booleanValue();
        boolean landscape = context.getBoolValue("http://www.ist-music.eu/Ontology_v0_1.xml#
                Concepts.Entities.Device.Screen; http://www.ist-music.eu/Ontology_v0_1.xml#
                Concepts.Scopes.Screen.Landscape ", false);
        // TODO: Translate the Pseudocode to source code
/*------------------------------------
            utility = 1.0; if landscape == landscapeProvided
        = 0.0, otherwise
------------------------------------*/
            //Code corresponding to the PseudoCode
//#BlockStart number=1 id=_iXAadoPGEduaib1rbQg5jQ_#_0
            utility = (landscape == landscapeProvided)? 1.0:0.0;
//#BlockEnd number=1
        return new Double(utility);
    }
}
```

Figure 46: Source code completion for the utility function

The automatically generated source code is manually enhanced by providing the calculation, as marked by TODO. The Pseudocode in the model is included in the generated source code within comments, while the developer can edit and add the code within a block so that the code is preserved, even when a new transformation of the model, possibly following some updates in the model, is made. The manually added source code is presented within the red rectangle.

The Component corresponding to the plan performs only the printing of corresponding messages on the console indicating if the landscape mode is set or not. The source code (hand written) is presented in Figure 47.

```
package unantadaptation.bundles.bundle1;
import org.istmusic.mw.adaptation.configuration.ConfigurableImpl;

public class UnanticipatedHelloWorldComponent extends ConfigurableImpl {
    boolean landscape = false;
    public void startActivity() {
        if (landscape){
            System.out.println("Hello, world!");
            System.out.println("Landscape");
        }
        else {
            System.out.println("Hello,");
            System.out.println("world!");
            System.out.println("Portrait");
        }
    }
    public void setLandscapeMode(Boolean landscape) {
        this.landscape = landscape.booleanValue();
    }
}
```

Figure 47: Source code of the UnanticipatedHelloWorldComponent component

8.1.2 Bundle 2

The second bundle contains only two plans; both of them would be matched as realizations to the application type presented in bundle 1. The plans corresponding to these components are presented in Figure 48 and Figure 49.

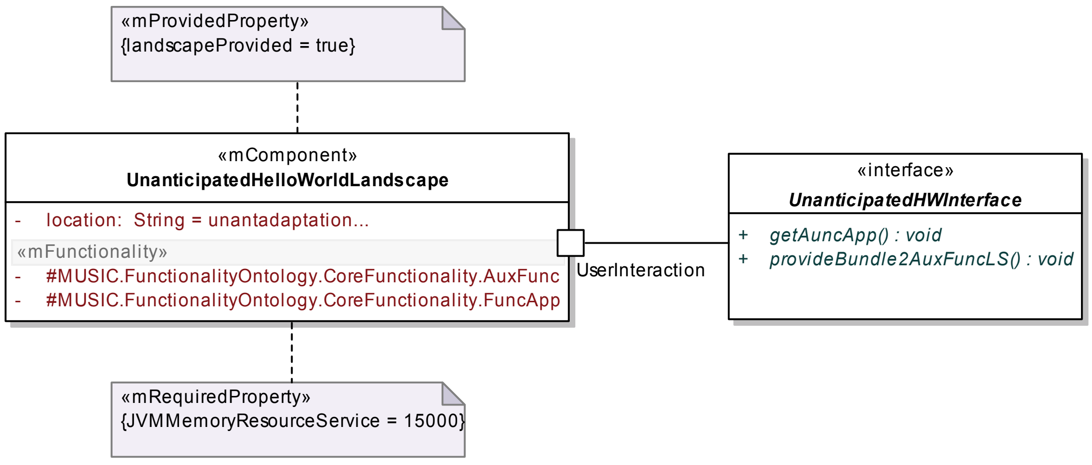

«mProvidedProperty»
{landscapeProvided = true}
«mComponent»
UnanticipatedHelloWorldLandscape
- location: String = unantadaptation...
«mFunctionality»
- #MUSIC.FunctionalityOntology.CoreFunctionality.AuxFunc
- #MUSIC.FunctionalityOntology.CoreFunctionality.FuncApp
UserInteraction
«interface»
UnanticipatedHWInterface
+ getAuncApp() : void
+ provideBundle2AuxFuncLS() : void
«mRequiredProperty»
{JVMMemoryResourceService = 15000}

Figure 48: Model of the UnanticipatedHelloWorldLandscape component

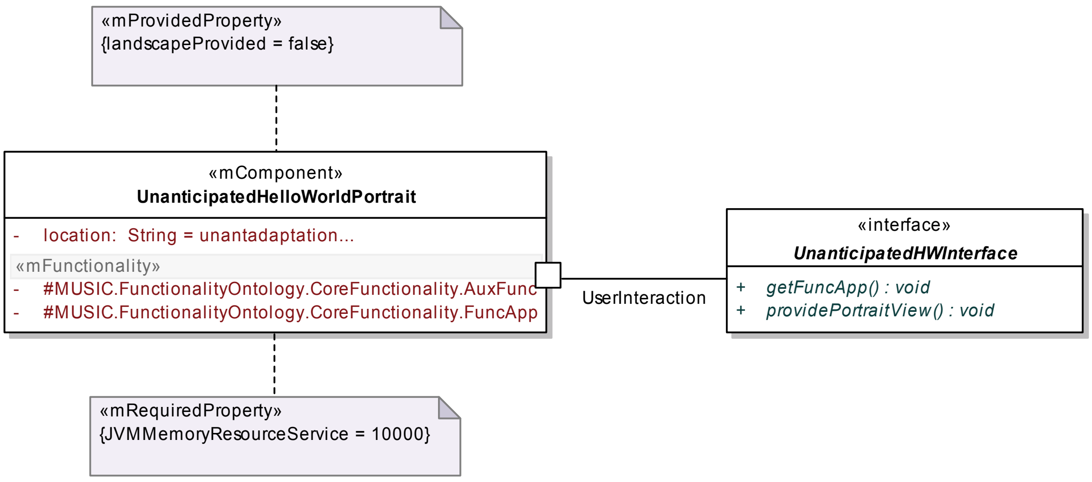

«mProvidedProperty»
{landscapeProvided = false}
«mComponent»
UnanticipatedHelloWorldPortrait
- location: String = unantadaptation...
«mFunctionality»
- #MUSIC.FunctionalityOntology.CoreFunctionality.AuxFunc
- #MUSIC.FunctionalityOntology.CoreFunctionality.FuncApp
UserInteraction
«interface»
UnanticipatedHWInterface
+ getFuncApp() : void
+ providePortraitView() : void
«mRequiredProperty»
{JVMMemoryResourceService = 10000}

Figure 49: Model of the UnanticipatedHelloWorldPortrait component

Please note that these plans do not contain any parameters; rather they are different plans with different sets of properties and resource requirements. The functionalities provided by them also differ; however, both of them realize the functionality required to realize the application type.

Each of these plans also contains a utility function, which is similar to the utility function presented in Figure 42, with a different name, of course.

8.1.3 Bundle 3

Bundle 3 is very similar to bundle 1. However, no application type or component type is contained in it. There is one atomic realization plan which uses parameters to differentiate between a landscape mode and a portrait mode. It also has one utility function.

8.1.4 Execution of the Test

This test does not involve any performance issue of the device because of the very small size of the application variability model. Therefore, it is performed on a laptop: IBM Thinkpad, X41, 1.5 GHz Pentium processor with 512MB RAM. The execution of the test involves a few steps as explained below:

1) The middleware bundles are started and the MUSIC GUI appears.

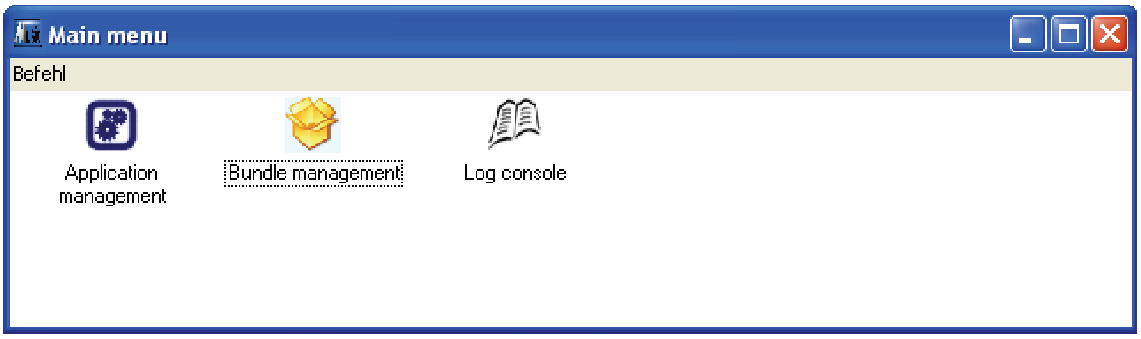


Figure 50: The MUSIC graphical user interface (GUI)

2) Using Bundle management of the GUI, bundles can be added or removed.

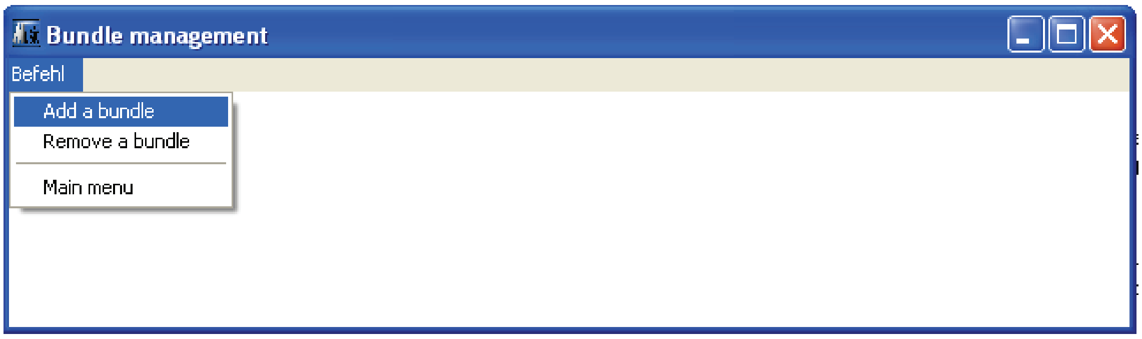


Figure 51: Adding a bundle using the GUI

3) Bundles are already created using the Eclipse plug-in Export wizard. We select the first bundle to install it.

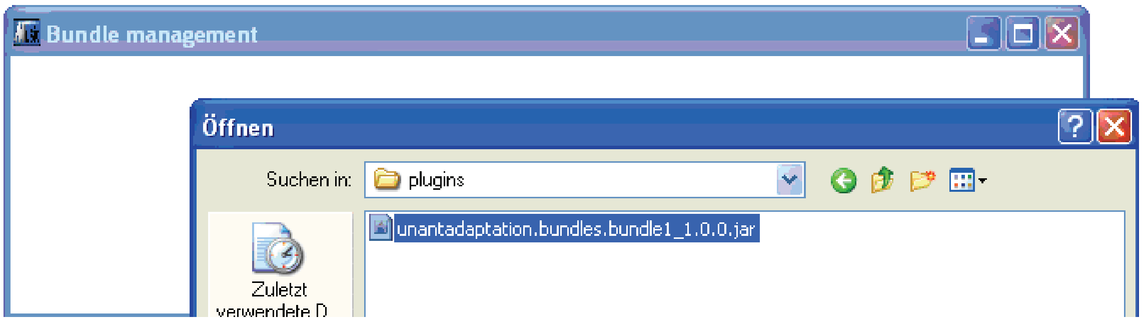
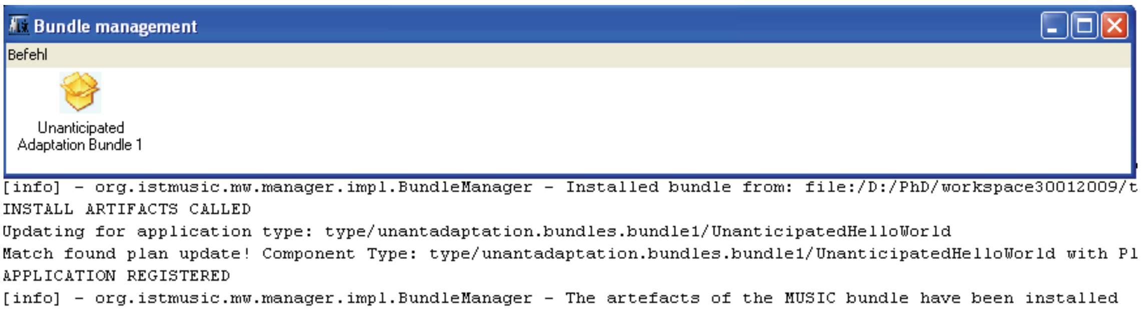


Figure 52: Bundles are selected from the created jars

4) After successful installation of the bundle, it appears on the GUI and some console output is presented.

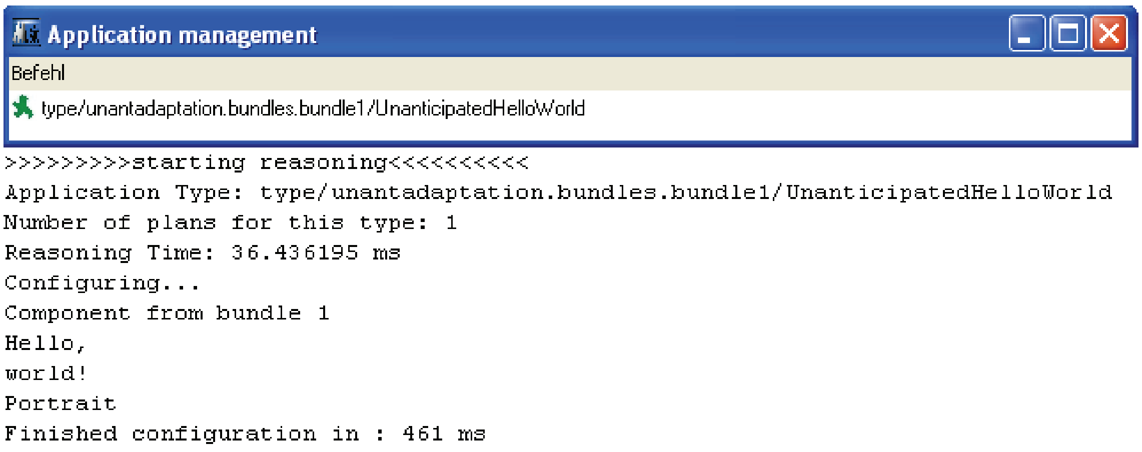


```
[info] - org.istmusic.mw.manager.impl.BundleManager - Installed bundle from: file:/D:/PhD/workspace30012009/t
INSTALL ARTIFACTS CALLED
Updating for application type: type/unantadaptation.bundles.bundle1/UnanticipatedHelloWorld
Match found plan update! Component Type: type/unantadaptation.bundles.bundle1/UnanticipatedHelloWorld with Pl
APPLICATION REGISTERED
[info] - org.istmusic.mw.manager.impl.BundleManager - The artefacts of the MUSIC bundle have been installed
```

Figure 53: Plans and types are matched during the bundle installation

The information on the console shows the location from where the bundle is installed. During the installation of the bundle artifacts, the component type, application type and plan repositories are updated. The matching between types and plans is done at this stage. When an application type is registered, it is indicated explicitly.

5) Using 'Application management' menu registered applications can be viewed on the GUI. The GUI also aids launching the application. The green icon indicates a successful configuration of the application.

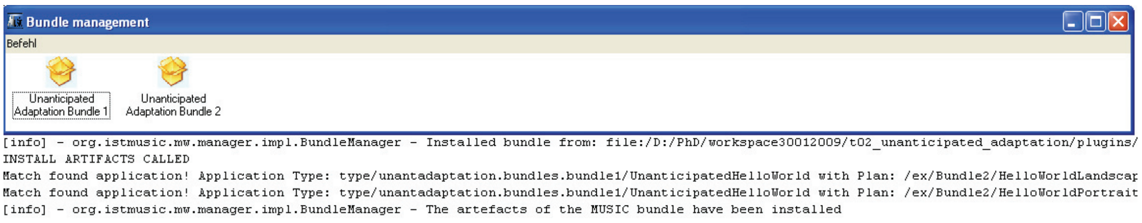

Application management
Befehl
type/unantadaptation.bundles.bundle1/UnanticipatedHelloWorld

```
>>>>>>>>>starting reasoning<<<<<<<<<
Application Type: type/unantadaptation.bundles.bundle1/UnanticipatedHelloWorld
Number of plans for this type: 1
Reasoning Time: 36.436195 ms
Configuring...
Component from bundle 1
Hello,
world!
Portrait
Finished configuration in : 461 ms
```

Figure 54: The application is launched using the GUI, and the console output is observed

With only the first bundle installed, there is only one plan available for the application. This is indicated by the information provided on the console. When the component is instantiated, it prints a few console messages to indicate which bundle is used to configure the application and also whether it is the landscape or the portrait mode. By using the ‘change orientation’ option, adaptations can be triggered and every time, it switches between Portrait and Landscape mode as will be evident also from the console output.

6) The second bundle is chosen to install. After a successful installation, it appears on the list of bundles. Also, the console output shows some information on the matching between plans and types.

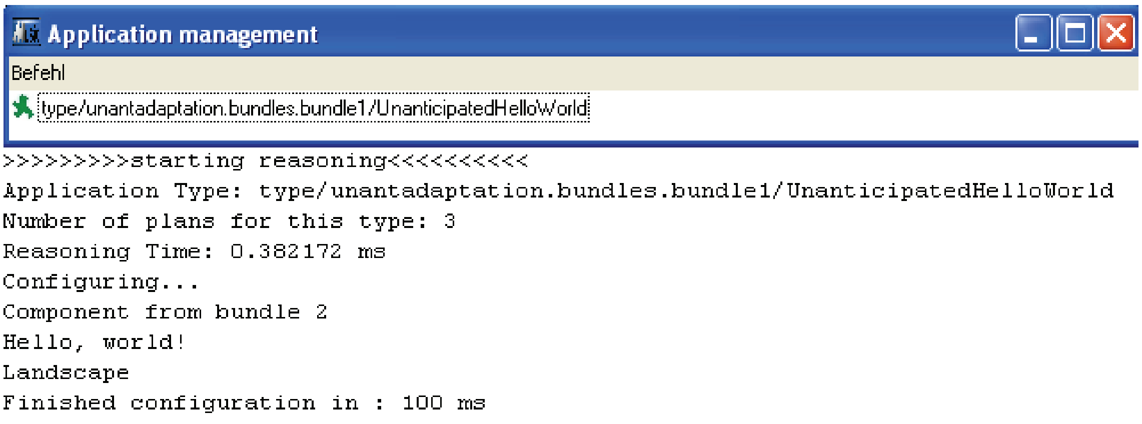

Bundle management
Befehl
Unanticipated Adaptation Bundle 1
Unanticipated Adaptation Bundle 2

```
[info] - org.istmusic.mw.manager.impl.BundleManager - Installed bundle from: file:/D:/PhD/workspace30012009/t02_unanticipated_adaptation/plugins/
INSTALL ARTIFACTS CALLED
Match found application! Application Type: type/unantadaptation.bundles.bundle1/UnanticipatedHelloWorld with Plan: /ex/Bundle2/HelloWorldLandscap
Match found application! Application Type: type/unantadaptation.bundles.bundle1/UnanticipatedHelloWorld with Plan: /ex/Bundle2/HelloWorldPortrait
[info] - org.istmusic.mw.manager.impl.BundleManager - The artefacts of the MUSIC bundle have been installed
```

Figure 55: The second bundle is installed

Here, note that the application type from bundle 1 is matched with the plans provided by bundle 2. Thus, artifacts from two independently developed bundles can co-operate through the matching process.

7) When adaptation is triggered as this situation, by using the ‘switch orientation’ menu, while the previous configuration of the application is still running, the console output indicates that currently there are three different plans – one from bundle 1 and two from bundle 2 - available to realize the application type. Using the adaptation reasoning process the plan providing the highest utility is selected to reconfigure the application.

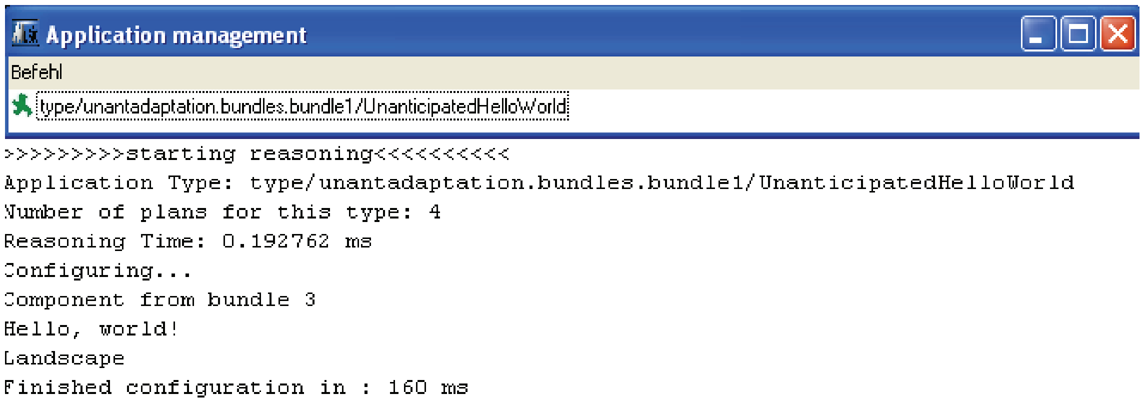

Application management
Befehl
type/unantadaptation.bundles.bundle1/UnanticipatedHelloWorld

```
>>>>>>>>>starting reasoning<<<<<<<<<
Application Type: type/unantadaptation.bundles.bundle1/UnanticipatedHelloWorld
Number of plans for this type: 3
Reasoning Time: 0.382172 ms
Configuring...
Component from bundle 2
Hello, world!
Landscape
Finished configuration in : 100 ms
```

Figure 56: Adaptation is triggered and the console output is observed

In this particular screenshot, the component is selected from bundle 2. However, it could be selected from bundle 1 as well. The utility function is slightly modified from what is presented in Figure 46. The marked line (*utility = (landscape == landscapeProvided)? 1.0:0.0;*) of that figure is replaced by '*utility = (landscape == landscapeProvided)? java.lang.Math.random():0.0;*' This is done to ensure that the components can be selected from any of the bundles based on the random value, while the wrong mode is discarded by setting utility to 0.0. That means, when Landscape mode of the GUI is selected, the component corresponding to that mode will be selected; but the selection of bundle depends on the random value.

8) Bundle 3 is deployed and the console output is observed by triggering adaptation with the help of changing the orientation of the GUI. Now, there are four different plans to realize the application.


Application management
Befehl
type/unantadaptation.bundles.bundle1/UnanticipatedHelloWorld

```
>>>>>>>>>starting reasoning<<<<<<<<<
Application Type: type/unantadaptation.bundles.bundle1/UnanticipatedHelloWorld
Number of plans for this type: 4
Reasoning Time: 0.192762 ms
Configuring...
Component from bundle 3
Hello, world!
Landscape
Finished configuration in : 160 ms
```

Figure 57: Console output after adaptation with all three bundles

8.1.5 *Comments on the Test Results*

Based on the test and observed results, as explained at different steps of section 8.1.4, we can deduct the following:

- The variability architecture of the application is created at runtime, detecting the deployment of new bundles and adding them automatically in the variability architecture by matching plans with types.

- The application does not need to be stopped to reconfigure using components provided by new bundles.
- The matching process is done automatically on the background and does not affect the application or adaptation reasoning.
- Bundles can be created by different developers, without prior knowledge of the application that will use the corresponding components.

8.2 Testing Scalability

This test addresses the scalability issue and evaluates the performance of the adaptation reasoning approach presented in section 5.3. We create two arbitrarily large variability models both containing one application type, named ‘LargeApplication’.

8.2.1 Variability Models under Test

The first variability model consists of 63 component types and 260 plans, while the second one introduces one more component type having 9 additional realization plans. The variability model is created completely arbitrarily as can be evident from a sample list of component types and plans, as presented in Table 4.

Table 4: Sample list of component types and realization plans (matched at runtime)

Type	**Plans**	**Number of plans**
LargeApplication	CR1, CR2, CR3, CR4, AR5	5
CT11	AR111, AR112, CR113, CR114, AR115, AR116	6
CT1131	AR11311, AR11312, AR11313	3
CT1141	AR11411, AR11412, AR11413, AR11414	4
CT12	AR121, AR122, AR123, AR124, CR125, CR126	6
CT1251	AR12511, AR12512, AR12513	3
CT1261	AR12611, AR12612	2
…	…	…
CT4541	AR45411, AR45412, AR45413	3

Clearly, it is not needed to present the details of the model. However, with a closer look at Table 4 will reveal that the ‘LargeApplication’ application type has four composite realization plans (annotated using CR) and an atomic realization plan (AR). Each of the composite plans has a composition; for example, the composition for the CR1 plan is presented in Figure 58 (without details of the port types, interfaces or functionalities).

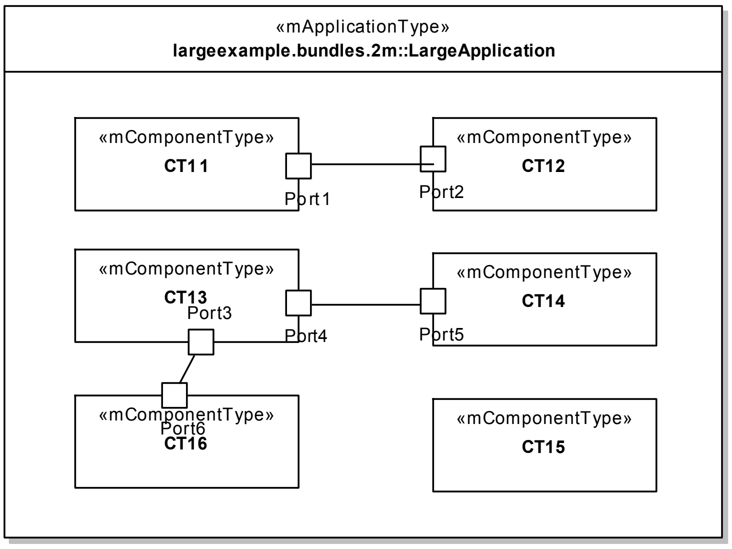


Figure 58: Composite structure of a realization of the LargeApplication application type

The composition has six component types, each of which has a number of plans, both atomic and composition plans. Each of these composition plans again has a composition of component types and so on. In order to calculate total number of variants, we can proceed from the bottom level of the variability model. For example, CR113 contains a single component CT1131 and CR114 contains only CT1141 in its combination. They have 3 and 4 plans respectively. Therefore, total number of plans for realizing CT11 is 3+4+(6-2) = 11.

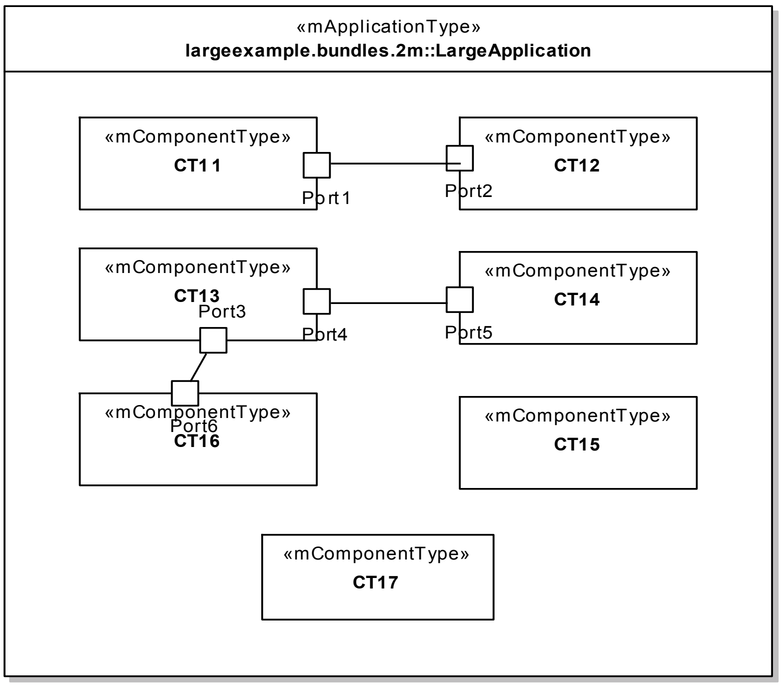


Figure 59: Composite structure of the plan introducing one more component type

Similarly, the number of plans for other component types in the composition of Figure 58 can be calculated. The total number of variants, corresponding to CR1 is a product of all these numbers, which in our experiment is 1,698,840. Similarly number of variants corresponding to CR2, CR3 and CR4 can be calculated. AR5 itself contributes to 1 variant. The number of application variants is obtained by adding these numbers. In our set up for the first variability model, we have a total of 2,004,697 variants.

The second variability model keeps everything unchanged, except introducing one more component type (CT17) in the composition as shown in Figure 59. Nine plans are added for this newly introduced component type. This results in multiplying the number of variants for CR1; i.e., 1,698,840 by 9 and therefore, the total number of application variants sharply increases to 15,595,417.

The model is transformed to generate source code corresponding to the adaptation capability of the application as well as component skeletons. The automatically generated source code for the adaptation capability must be enhanced by filling out the utility functions. The component skeletons also need to be manually enhanced implementing the functionalities supposed to be provided by the component. However, for this test, we have customized the transformation script to automatically generate utility functions for atomic plans so that the utility value is assigned as a random number between 0.0 and 1.0 (using the *java.lang.Math.random()* method). Utility functions are also generated automatically for composite realization plans using the weight information for each of the component types. The component skeletons were not manually enhanced by adding the implementations for functionalities; rather, the constructor will print a console output when a particular component is instantiated. This way, we can trace the selected variant of the application.

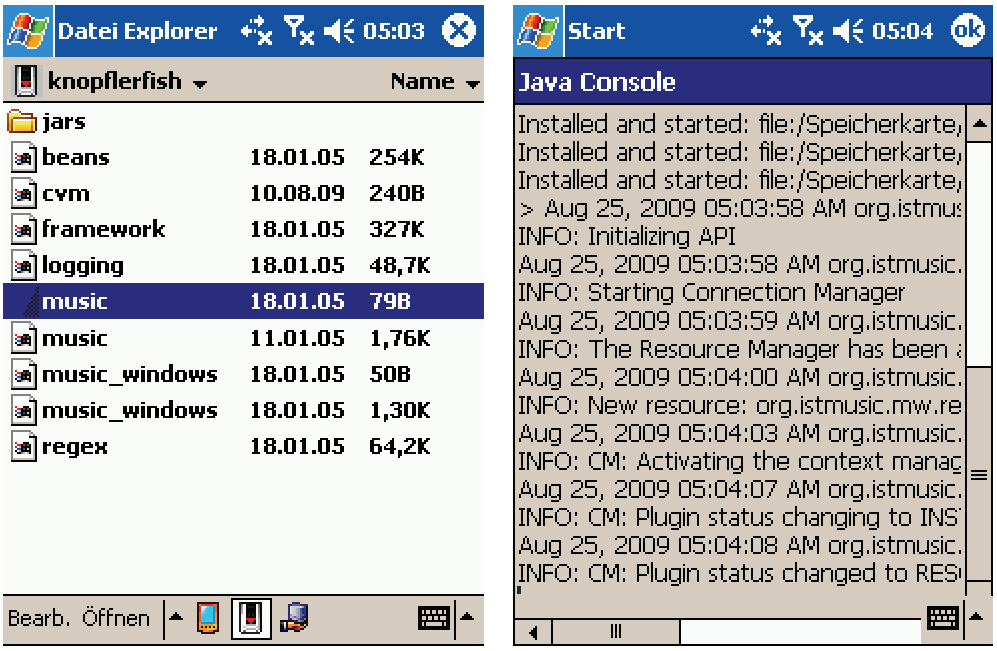
8.2.2 Execution of the Test

The test is performed on a PDA, HP iPAQ 6340 Pocket PC, TI OMAP1510 Processor, 56MB RAM, running Windows Mobile 2003. We have used PhoneME [84], which is a fully featured open source JVM with knopflerfish [85] OSGi R4, which is an open source OSGi Service Platform. The reason behind choosing a PDA is quite obvious. In this particular test, we investigate the performance of the adaptation reasoning approach and therefore, a PDA, being resource critical, becomes an automatic choice over a laptop. If the adaptation reasoning of the variability model performs well on a PDA, we can safely say that it would perform faster on a laptop or on a Desktop PC.

The execution of the test consists of the following sequential steps:

1) The middleware is started using a link (music.lnk[19]) to the cvm.exe file along with a set of arguments (cvm.args). For this test both the middleware bundles and the single application bundle are automatically installed, as it will be shown by some Java console output.

[19] We keep the configuration files unchanged (of course, updating the list of bundles to load), as they have been for testing MUSIC applications.



jars		
beans	18.01.05	254K
cvm	10.08.09	240B
framework	18.01.05	327K
logging	18.01.05	48,7K
music	18.01.05	79B
music	11.01.05	1,76K
music_windows	18.01.05	50B
music_windows	18.01.05	1,30K
regex	18.01.05	64,2K

```
Installed and started: file:/Speicherkarte,
Installed and started: file:/Speicherkarte,
Installed and started: file:/Speicherkarte,
> Aug 25, 2009 05:03:58 AM org.istmus
INFO: Initializing API
Aug 25, 2009 05:03:58 AM org.istmusic.
INFO: Starting Connection Manager
Aug 25, 2009 05:03:59 AM org.istmusic.
INFO: The Resource Manager has been a
Aug 25, 2009 05:04:00 AM org.istmusic.
INFO: New resource: org.istmusic.mw.re
Aug 25, 2009 05:04:03 AM org.istmusic.
INFO: CM: Activating the context manag
Aug 25, 2009 05:04:07 AM org.istmusic.
INFO: CM: Plugin status changing to INS
Aug 25, 2009 05:04:08 AM org.istmusic.
INFO: CM: Plugin status changed to RES
```

Figure 60: Starting the middleware

2) The MUSIC GUI can be used for managing bundles, applications etc. as well as for visualizing the information on the Log console.

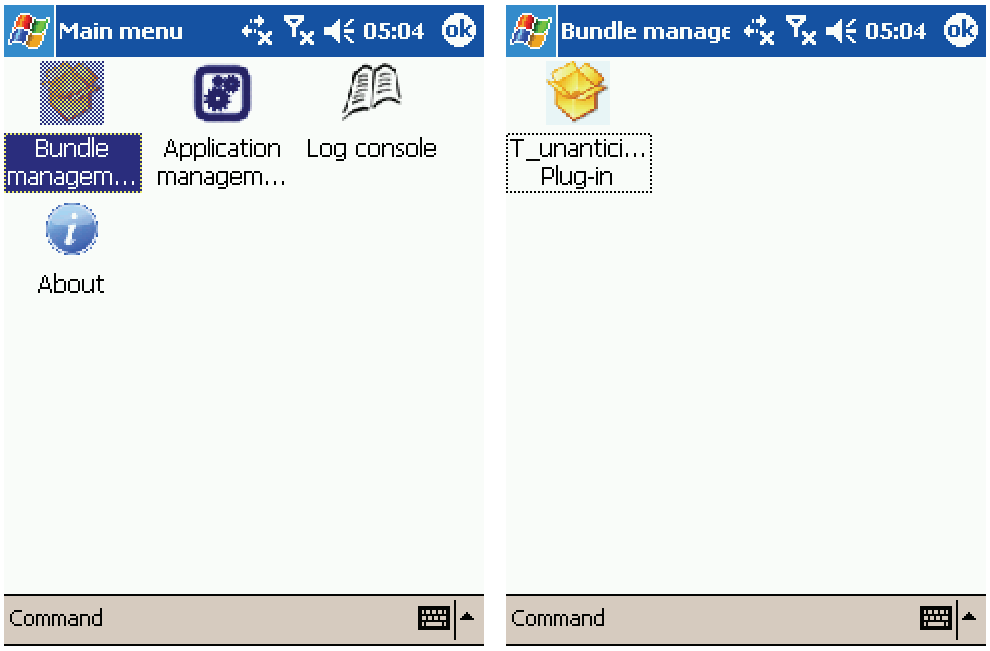
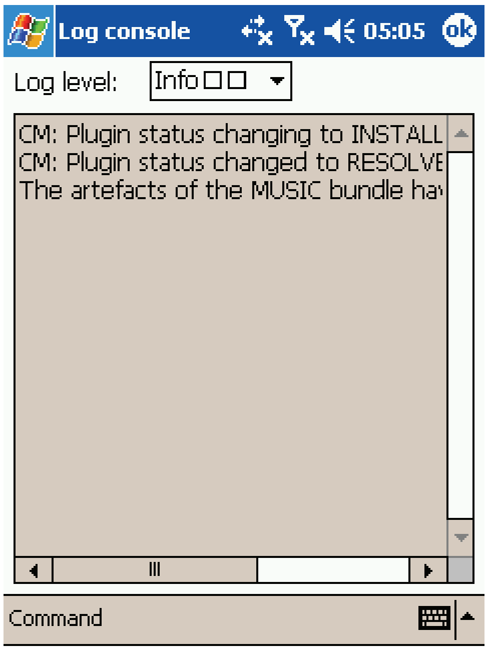


Figure 61: The MUSIC graphical user interface on a mobile device

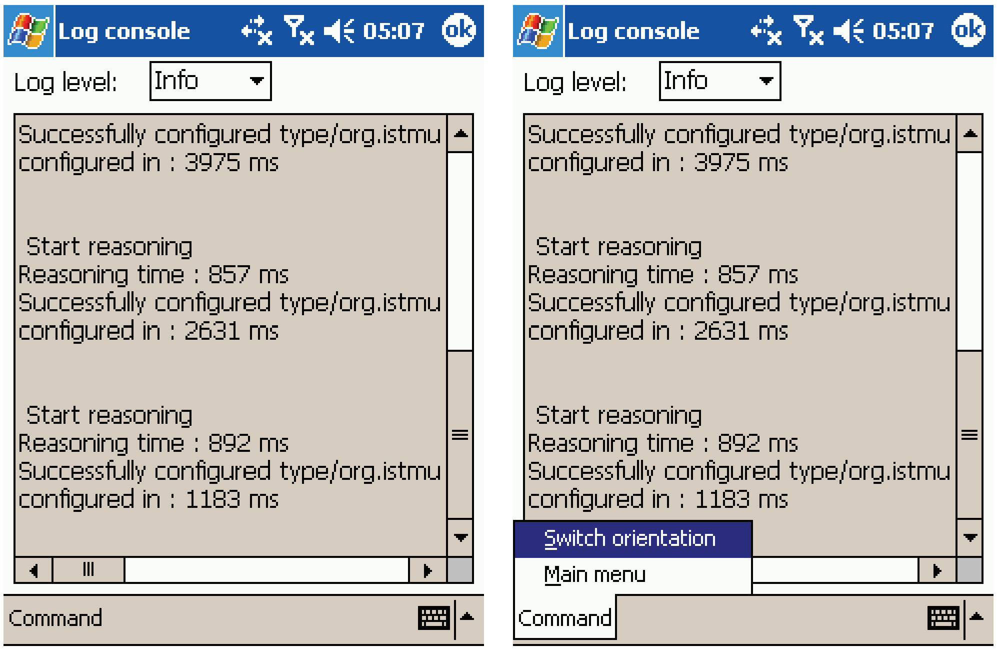
3) The application bundle is installed automatically along with the middleware bundles. For this particular case, the installation takes almost a minute because it has to match the application type and all the component types with all the plans available in the bundle. Moreover, it prints a lot of (Java) console messages to provide information on the matched types and plans. In actual application, the process will be faster, especially because no such console output is required. The Log console supports viewing only selected information and using it we can see if the installation is complete.



```
CM: Plugin status changing to INSTALL
CM: Plugin status changed to RESOLVE
The artefacts of the MUSIC bundle ha
```

Figure 62: Log Console indicates the installation of the bundle

4) Using the GUI, the application is launched and the output on the Log console is observed. For this case, we only observe the reasoning time[20] and configuration time[21] after each successful reconfiguration of the application following an adaptation process.



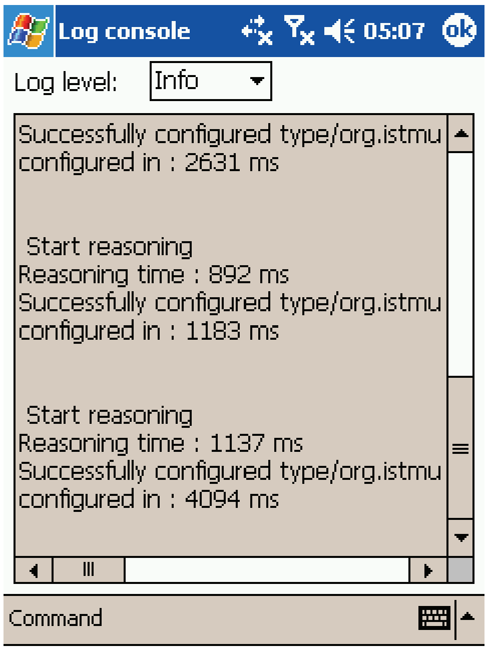
```
Successfully configured type/org.istmu
configured in : 3975 ms


 Start reasoning
Reasoning time : 857 ms
Successfully configured type/org.istmu
configured in : 2631 ms


 Start reasoning
Reasoning time : 892 ms
Successfully configured type/org.istmu
configured in : 1183 ms
```

Figure 63: Output log showing the adaptation reasoning and configuration time

[20] Reasoning time is calculated as the interval between the start of the adaptation reasoning, following the detection of a context change, and selection of the best-fit application variant.

[21] Configuration time is the interval between the selection of the best-fit variant and changing the application configuration to this selected one.

5) The middleware provides sensors to detect, if the landscape or the portrait screen is chosen. This is used to trigger adaptation by changing the orientation of the screen. To trigger a new adaptation, the ‘Switch orientation’ menu is tapped.



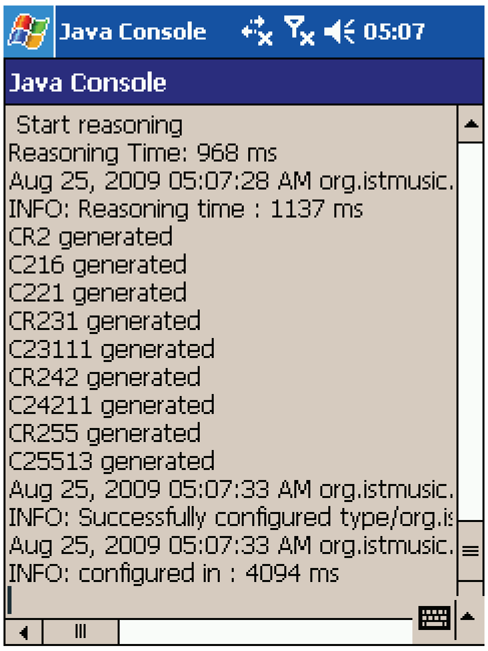
Log level: Info

```
Successfully configured type/org.istmu
configured in : 2631 ms


 Start reasoning
Reasoning time : 892 ms
Successfully configured type/org.istmu
configured in : 1183 ms


 Start reasoning
Reasoning time : 1137 ms
Successfully configured type/org.istmu
configured in : 4094 ms
```

Figure 64: Select switch orientation to trigger a new adaptation

6) By switching the orientation, adaptation reasoning time and configuration time are recorded in order to calculate an average value. For this particular test, we have recorded those numbers for only 20 times (see Table 5). The reason is that there is no big deviation in the adaptation reasoning time (except the first reading) and therefore, even such a few numbers of readings would be a correct representative of an average reasoning time.

7) After closing the middleware GUI, the Java console output can be observed to see messages on the selected configuration following each adaptation.



```
 Start reasoning
Reasoning Time: 968 ms
Aug 25, 2009 05:07:28 AM org.istmusic.
INFO: Reasoning time : 1137 ms
CR2 generated
C216 generated
C221 generated
CR231 generated
C23111 generated
CR242 generated
C24211 generated
CR255 generated
C25513 generated
Aug 25, 2009 05:07:33 AM org.istmusic.
INFO: Successfully configured type/org.is
Aug 25, 2009 05:07:33 AM org.istmusic.
INFO: configured in : 4094 ms
```

Figure 65: Java console presents message when a component is instantiated

After finishing the test with the first variability architecture, the second one checked exactly the same way.

8.2.3 Test Results and Comments

In Table 5 we present the adaptation reasoning time and (re)configuration time corresponding to each new adaptation.

Table 5: Evaluation results (on HP iPAQ 6340 Pocket PC, TI OMAP1510 Processor, 56MB RAM, Windows Mobile 2003, running PhoneME + knopflerfish)

Adaptation No.	**Reasoning time (ms)**		**(Re)configuration time (ms)**	
	Model 1: 2,004,697 variants	*Model 2: 15,595,417 variants*	*Model 1: 2,004,697 variants*	*Model 2: 15,595,417 variants*
1.	1832/803	3800/2019	1467/4520	1615/7715
2.	885	1793	2871	7912
3.	864	2421	3262	5208
4.	807	2545	3341	7631
5.	818	2378	1126	6234
6.	855	1806	4179	8372
7.	824	1993	5590	5191
8.	810	1875	3603	1592

9.	818	2730	4276	6169
10.	848	1783	2701	6134
11.	1203	2167	4011	6232
12.	815	1900	3742	4233
13.	853	1949	4383	5887
14.	1184	1975	4384	7721
15.	859	2511	3664	10417
16.	818	1749	4522	7020
17.	879	1875	4102	1344
18.	1199	1942	4082	6377
19.	874	1836	4806	5729
20.	866	1897	4319	5981
Average	**894.1**	**1959.75**	**3874.2**	**6154.95**

Based on the evaluation result of Table 5, the following remarks can be made:

- The adaptation reasoning time employing the developed adaptation reasoning approach (see section 5.3) is quite within the acceptable limit, even for huge number of application variants.
- The reasoning time is not influenced drastically with the increase in the number of possible application variants. With almost 8 times increase in the number of application variants, the reasoning time is only doubled.
- The configuration time does not depend on the number application variants; rather it depends on the number of components to instantiate, or more specifically, it is the time required to switch from the older configuration to the newly chosen one. So, it depends on the difference between these two configurations. The average reconfiguration time is slightly increased, by a factor of 1.6, which can be explained by the fact that one new component is introduced in one of the configuration. The time required for initial configuration is quite low, because in this case the middleware does not need to deactivate components of the old configuration (there is none in this case) that is not required for the new configuration. A few other reconfiguration times are also low. We have checked that this happens when the older configuration contains the single component corresponding to AR5.
- The overall adaptation time, adding up the time required for the detection of context changes, reasoning of adaptation and reconfiguration of the application is still within a few seconds, which is quite acceptable.

- It is to note that for the initial adaptation, there are two numbers. This is a problem with the middleware that the adaptation process runs twice when an application is started and, depending on the case, the application is configured twice. When an application is started from the GUI, this sends an event which triggers the adaptation process. In this process, the application registers its context dependencies. If the application needs a context sensor that hasn't been yet activated, the context manager queues the activation of the sensor. The problem is that, given that sensor activation is not synchronous, the adaptation process continues before the context sensors are activated. So, when the utility function is evaluated, the context queries return the default value specified in the utility function, as the context elements are not yet available. For that reason, the first adaptation process is done without the real context values. After some time, the context manager processes the queue events and activates the context sensors. Then, the sensors initialize the correct context values. The adaptation process is triggered by these context change events, evaluating again the utility function, this time with the correct context values.

- In order to compare the result with the MUSIC solution, we have run an equivalent MUSIC variability architecture corresponding to the first variability model on the MUSIC middleware (v0.2.2). It takes about 14 minutes on a Desktop PC running Windows XP with Pentium4 3GHz processor and 1GB RAM. Such huge difference in adaptation reasoning is quite obvious from the fact that in the older approaches all the application variants considered separately (millions of combinations) , while in the approach we have presented evaluates utility for the plans (a few hundred only).

- Theoretically, as mentioned in section 5.3.4, the complexity of the adaptation reasoning approach with respect to the number of plans can be expressed as O(n), while that for a reasoning approach considering each application variant separately is $O(n^c)$, where c is the number of component types in the composition.

9 Discussions

Working in the area of context awareness and self-adaptation motivates us to vision about more intelligent systems that can ‘think’ ahead of their developers. Human being can most often behave intelligently in new and possibly unforeseen situations, utilizing the support ‘at hand’ at that particular context. Following a similar thought process, we have worked on providing support of the unanticipated adaptation to mobile applications. Clearly, the meaning of unanticipation itself has some limitations and the support provided in this thesis, by no means, solves all the challenges related to the unanticipated adaptation. However, we consider the solution as one step forward towards the direction of providing such support, while future researches can only improve it.

In the following, we discuss the limits of the unanticipation concept in terms of adapting mobile applications. Afterwards, we discuss the extent to which we have provided a solution to it, pointing to the shortcomings of the solution and possible improvements in the future.

9.1 Limits of Unanticipation

In theory, all adaptations must remain unanticipated until some point [5]. Therefore, different people use the term ‘unanticipated adaptation’ for slightly different meanings. A popular understanding of ‘unanticipated’ is that ‘which has not been foreseen at design time’ [6][7]. Therefore, ‘unanticipated’ software adaptation can also be understood to mean software adaptations that are not anticipated until the execution of that software is started [8]. Unfortunately, for mobile applications running on a distributed environment with the ability to use services and components provided by others in the adaptation domain, not all needs for adaptation can be foreseen even at the deployment time or when the software has started. Because, at runtime the context may change, introducing a change in the available services and devices and therefore, a proper adaptation decision should be based on the ‘situation at hand’. In this work, we view the unanticipated adaptation till the extent that the adaptation remains unanticipated till the point of adaptation reasoning. The aim of adopting such view is to facilitate the realization of a user’s application by components and services from other independent users and/or service providers available during the adaptation reasoning.

The solution to address unanticipated adaptation can not always be based on particular scenarios; rather it should be generic, as much as possible, in order to cover ‘any’ situation in the ideal case. However, finding a generic solution, which is flexible in that extent, is quite challenging, if not impossible at the current state of the art. Therefore, a practical solution is limited by several factors. For example, in this thesis, our support of unanticipated use of components is limited to those cases, where the developed components are compliant to the U-MUSIC information model. We also support integration of third party services through a number of discovery and communication protocols. However, it is limited by the support for the number of discovery and communication protocols and it certainly does not cover all services that may be available in the service landscape of a ubiquitous environment.

A solution to the unanticipated adaptation must be meaningful from a user's point of view. This integrates users' preferences in the adaptation decision. In the case of the unanticipated adaptation, this can not be always foreseen and therefore, the user may perceive an adaptation that he does not like. Such problems should be solved as much as possible.

9.2 Support of Unanticipation

The work presented in this thesis is based on the results obtained in the MADAM and the MUSIC projects. In MUSIC, there is some on-going research on reasoning about uncertain context information. However, that topic deals with providing adaptation solution, even when there is some ambiguity in the context information and the unanticipated adaptation problem, as it is defined in section 1.1.3, is not explicitly addressed. In this work, we extend the MUSIC solution by introducing the unanticipated adaptation in the sense that applications from independent developers, having the U-MUSIC middleware as the common understanding point, can interact and benefit from each others development. Devices using such applications can come across to each other in a completely unanticipated manner. Compared to MUSIC, this gives us the advantage that a particular component is no longer bound to realizing a particular component type only. It can be used to realize any component type requiring only a subset of functionalities offered by the component. This also facilitates imprecise matching, when a component can realize a component type only approximately. Such usage gains advantage in the case of the unanticipated adaptation, especially when no perfectly matching component is available.

In that direction, from our work in the MUSIC project we have presented the integration of third-party services in the application configuration. However, the integration of services can be partially anticipated, because the need for such services has to be estimated in some extent at design time.

As an extension to the MUSIC solution, we have also provided the support for unanticipated adaptation, facilitating the use of components from 'independent' developers in configuring the application at runtime. We have developed and updated MUSIC concepts, as necessary, provided mechanisms to dynamically match application components and their meta-information, adapt the application in quick time using a new adaptation reasoning approach. We have also provided an updated methodology that any application developer needs to follow in order to develop unanticipated adaptive applications. The concepts and the methodology allow individual developers to focus on his development, without worrying about what the others are developing. With the aid of an initial implementation (middleware) of the conceptual development, we have used arbitrary applications to demonstrate the adaptation in an unanticipated way.

Adaptive mobile applications, in general, suffer from the inability of providing an adaptation solution, which is quick enough to cope with the highly dynamic environment that they are operating on. Our adaptation reasoning approach provides a solution to this problem. The specification of the variability model has become easier for developers, because they need to focus on their components only. Because of the need to focus only on smaller areas of the complete system/application, we also claim that the specification of utility functions and property evaluators has also been eased. With the help of two variability models, creating millions of application variants, we have tested the effectiveness of the solution.

Comparing to the challenges in supporting unanticipated dynamic adaptation, as presented in section 1.4, the creation of application variability at runtime is very-well supported. The heterogeneity aspect is supported with some limitations that we will discuss in the next section. Dynamic discovery of devices and services is supported for particular discovery and communication protocols. Context-sensing and reasoning is supported well (from MUSIC). We introduce and provide initial concepts for dynamic updates of requirements through runtime matching of plans and types corresponding to those requirements. Another important contribution of the work is in the area of adaptation reasoning. We have provided a solution that is not vulnerable to the scalability problem and can provide a very quick adaptation reasoning. We also provide partial support (ongoing work in MUSIC) for testing and validation.

9.3 Shortcomings

The solution provided in this work does not completely solve all the challenges introduced in section 1.4; rather it has a number of limitations. Introduction of the term 'functionality' improves the probability of using components from unknown developers to realize an application. However, it still suffers from the differences in developers thinking. Matching functionalities from independent developers is not an easy task and although we address that problem through introducing the option of using a functionality ontology, combining two ontologies to identify similar terms is still a research issue.

Our adaptation reasoning approach is based on four assumptions and therefore, it is as good as the validity of those assumptions. In order to gain reasoning speed, we have compromised facts that the choice of a particular component in the composition may influence the utility of another component. Also, in the case of perfect unanticipation, the developers may not always be able to provide a utility function for their components to fit in all possible situations. This requires that the utility function itself needs to be dynamic. This is probably not a shortcoming, because the specification of the utility functions is open, as long as they do not violate the assumptions. However, we did not discuss such complex case of utility functions in this thesis.

The current middleware implementation does not support all the concepts. The runtime matching of types and plans only consider functionalities. Moreover, we have implemented only the string matching support and the support for imprecise matching is not implemented in the middleware. The adaptation reasoning approach lacks the implementation of architectural constraints. We did not implement the introduction of new functionalities at runtime by the user. This requires an update to the MUSIC GUI so that a user can add functionalities through the GUI.

Although we have presented a scenario, related to real-life applications, we could not demonstrate it in this work for practical limitations.

9.4 Future Work

We have provided a solution in this thesis with the aim of stepping forward in the challenging world of the unanticipated adaptation. We have clearly identified the problems; but we could not fully address all of them. Therefore, there is a huge scope of improvements, from both research and development point of view, in the concerned area.

Based on our current status, the first task is to enhance the middleware implementation with a complete support for the already addressed concepts. An effective matching technique is still an interesting research topic, along with the tool support for merging Ontologies and identifying similar terms. The presented Ontologies for services and functionalities are also in their infancies and we are working on enhancing them to cover a rich set of related entities and concepts.

In this work, we have not addressed explicitly the Robustness and Security aspects. Our focus has been on the adaptation aspect of the application. However, these aspects are particularly important in a ubiquitous computing environment. Therefore, they must be addressed in order to apply the solution in practical applications.

Fortunately, we are still working on the MUSIC project, which will address some of the non-addressed challenges like security and robustness in some extent along with improving the existing solution. In MUSIC, we are also developing a number of trial applications to demonstrate context awareness and self-adaptation. Currently, in those demonstrations it is not planned to address the unanticipated adaptation, as presented in this work; but they can be useful to verify many aspects of adaptation, in general.

References

[1] Mobility and Adaptation–enabling Middleware (MADAM), project homepage: http://www.intermedia.uio.no/display/madam/Home (accessed on 03.11.2009)

[2] Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC), http://www.ist-music.eu (accessed on 03.11.2009)

[3] Philip. K. McKinley, Seyed M. Sadjadi, Eric P. Kasten and Betty H. C. Cheng, *A Taxonomy of Compositional Adaptation,* Tech report, Software Engineering and Network Systems Laboratory, Michigan State University. 2004.

[4] Roland Reichle, Mohammad U. Khan, and Kurt Geihs, *How to Combine Parameter and Compositional Adaptation in the Modeling of Self-Adaptive Applications*, PIK (Praxis der Informationsverarbeitung und Kommunikation) special issue on modeling of self-organizing systems, vol. 31, no. 1, pp. 34–38, 2008.

[5] Günter Kniesel, Joost Noppen, Tom Mens and Jim Buckley, *Unanticipated Software Evolution,* in ECOOP 2002 Workshop Reader, Malaga, Spain. Springer-Verlag (LNCS 2548), 2002.

[6] Kai-Uwe Mätzel and Peter Schnorf, *Dynamic Component Adaptation,* Ubilab Technical Report 97.6.1, Union Bank of Switzerland, Zurich, Switzerland. 1997.

[7] Jim Buckley, Tom Mens, Matthius Zenger, Awais Rashid and Gunter Kniesel, *Towards a Taxonomy of Software Change*, Journal of Software Maintenance and Evolution: Research and Practice (Special Issue on Unanticipated Software Evolution). Vol. 7, Issue 5, Pages 309 – 332, September, 2005.

[8] Barry Redmond, *Supporting Unanticipated Dynamic Adaptation of Object-Oriented Software*, Ph.D. Thesis, in Department of Computer Science, Trinity College Dublin, Dublin, 2003.

[9] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart and Rajarshi Das, *Utility Functions in Autonomic Systems,* In proceedings of First International Conference on Autonomic Computing (ICAC'04). 2004. p. 70-77.

[10] Mobility and Adaptation–enabling Middleware (MADAM), Deliverable *D2.2 Theory of Adaptation*, Editor – Jacqueline Floch, http://www.intermedia.uio.no/display/madam/D2.2+-+Theory+of+Adaptation, (accessed on 03.11.2009)

[11] Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC), Deliverable *D1.3 Intermediate Research Results on Mechanisms and Planning Algorithms for Self-adaptation*, Editor – Romain Rouvoy.

[12] Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC), Deliverable *D2.2 Initial Research Results on Methods, Languages, Algorithms and Tools to Modeling and Management of Context*, Editor - Massimo Valla, http://www.ist-music.eu/MUSIC/results/music-deliverables/docs/D2.2.pdf (accessed on 26.08.2009)

[13] Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC), Deliverable *D2.3 Intermediate Research Results on Methods, Languages, Algorithms and Tools to Modeling and Management of Context,* Editor – Nearchos Paspallis.

[14] Matthias Baldauf, Schahram Dustdar and Florian Rosenberg, *A Survey on Context-aware Systems,* International Journal of Ad Hoc and Ubiquitous Computing, Vol. 2, No. 4, pp.263–277, year 2007.

[15] Tao Gu, Hung K. Pung, and Da Q. Zhang, *A Middleware for Building Context-Aware Mobile Services*, 59th Vehicular Technology Conference (VTC '04), IEEE, Milan, Italy, May 2004.

[16] Anand Ranganathan, Roy H. Campbell, *A Middleware for Context-Aware Agents in Ubiquitous Computing Environments*, ACM/IFIP/USENIX International Middleware Conference, pp. 143-161, Rio de Janeiro, Brazil, June 2003.

[17] Ian Horrocks, *DAML+OIL: A Reason-able Web Ontology Language*, 8th International Conference on Extending Database Technology (EDBT), pp. 2-13, Prague, March 2002.

[18] Amir Padovitz, Seng W. Loke, and Arkady Zaslavsky, *The ECORA Framework: A Hybrid Architecture for Context-oriented Pervasive Computing*, Pervasive and Mobile Computing, Volume 4, Issue 2, pp.182-215, April 2008.

[19] Karen Henricksen and Jadwiga Indulska, *A Software Engineering Framework for Context-Aware Pervasive Computing*, 2nd IEEE International Conference on Pervasive Computing and Communications, IEEE Computer Society, pp. 77–86, 2004.

[20] Bob Hardian, *Middleware Support for Transparency and User Control in Context-aware Systems*, 3rd international Middleware Doctoral Symposium (MDS '06), Melbourne, Australia, November 27 - December 01, 2006, Vol. 185, ACM Press.

[21] Stephen S. Yau, Fariaz Karim, Yu Wang, Bin Wang and Sandeep K. S. Gupta, *Reconfigurable Context-Sensitive Middleware for Pervasive Computing*, IEEE Pervasive Computing, Vol. 1, No. 3, pp. 33-40, 2002.

[22] Anind K. Dey, *Providing Architectural Support for Building Context-Aware Applications*, PhD Thesis, College of Computing, Georgia Institute of Technology, 2000.

[23] Alan Newberger, Anind K. Dey, *Designer Support for Context Monitoring and Control*, Intel Research Berkeley, 2003.

[24] Davy Preuveneers and Yolande Berbers, *Adaptive Context Management Using a Component-based Approach,* In 5th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), vol. 3543 of LNCS, pp. 14–26, Athens, Greece, June 2005.

[25] Thomas Hofer, Wieland Schwinger, Mario Pichler, Gerhard Leonhartsberger and Josef Altmann, *Context-Awareness on Mobile Devices – the Hydrogen Approach*, In Proc. 36th Annual Hawaii International Conf. on System Sciences, pp.292-302, January 2003.

[26] Miguel A. Muñoz, Marcela Rodríguez, Jesus Favela, Ana I. Martinez-Garcia and Victor M. Gonzalez, *Context-aware Mobile Communication in Hospitals*, Computer, IEEE Computer Society, vol. 36, no. 9, pp. 38-46, Sept. 2003.

[27] Federica Paganelli, Gabriele Bianchi and Dino Giuli, *A Context Model for Context-Aware System Design Towards the Ambient Intelligence Vision: Experiences in the eTourism Domain,* In LNCS - Universal Access in Ambient Intelligence Environments,

Lecture Notes in Computer Science, Springer Verlag, Vol. 4397, pp. 173-191, August 2007.

[28] Ricardo C. A. da Rocha and Markus Endler, *Context Management in Heterogeneous, Evolving Ubiquitous Environments,* IEEE Distributed Systems Online, vol. 7, no. 4, IEEE Computer Society, April 2004.

[29] Hana K. Rubinsztejn, Markus Endler, Vagner Sacramento, Kleder Gonçalves and Fernando Nascimento, *Support for Context-aware Collaboration,* In LNCS - Mobility Aware Technologies and Applications, Springer Verlag, vol. 3284, pp. 37-47, 2004.

[30] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum and Alexander L. Wolf, *Architecture-based Approach to Self-adaptive Software,* IEEE Intelligent Systems and Their Applications, vol. 14, no. 3, p. 54-62, May 1999.

[31] Arun Mukhija, *CASA - A Framework for Dynamic Adaptive Applications,* Doctoral Thesis, University of Zurich, 2007.

[32] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl and Peter Steenkiste, *Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure*, IEEE Computer, vol. 37, no. 10, pp. 46-54, Oct 2004.

[33] João P. Sousa and David Garlan, *Aura: An Architecture Framework for User Mobility in Ubiquitous Computing Environment,* In proceedings of 3rd working IEEE/IFIP Conference on Software Architecture, Montreal, Canada, 2002.

[34] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani, *Recursive and Dynamic Software Composition with Sharing*, In proceedings of Seventh International Workshop on Component-Oriented Programming (WCOP02), Malaga, Spain, 2002.

[35] Stefanos Zachariadis, Cecilia Mascolo and Wolfgang Emmerich, *SATIN: A Component Model for Mobile Self Organisation*, In proceedings of Proc. of CoopIS, DOA and ODBASE. 2004. Agia Napa, Cyprus.

[36] Licia Capra, Wolfgang Emmerich and Cecilia Mascolo, *CARISMA: Context-Aware Reflective Middleware System for Mobile Applications,* IEEE Transactions on Software Engineering, vol. 29, no. 10, pp. 929-945, 2003.

[37] Pierre-Guillaume Raverdy and Rodger Lea, *DART: A Distributed Adaptive Runtime,* In proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98), 1998.

[38] Pierre-Charles David and Thomas Ledoux, *Towards a Framework for Self-adaptive Component-based Applications*, In proceedings of Distributed Applications and Interoperable Systems (DAIS'2003), Paris, France, 2003.

[39] John Keeney and Vinny Cahill, *Chisel: A Policy-driven, Context-aware, Dynamic Adaptation Framework,* In proceedings of Proc. of the 4th International Workshop on Policies for Distributed Systems and Networks, Lake Como, Italy, 2003.

[40] Manuel Oriol, *An Approach to the Dynamic Evolution of Software Systems*, PhD Thesis No. 556, University of Geneve, 2004.

[41] Carlos A. Flores-Cortés, Gordon S. Blair, Paul Grace, *An Adaptive Middleware to Overcome Service Discovery Heterogeneity in Mobile Ad Hoc Environments,* IEEE Distributed Systems Online, vol. 8, no. 7, art. no. 0707-o7001, July 2007.

[42] Holger Mügge, Tobias Rho, Daniel Speicher, Pascal Bihler, and Armin B. Cremers, *Programming for Context-based Adaptability Lessons learned about OOP, SOA, and AOP*, SAKS Woskshop, March 2007.

[43] Marcel Cremene, Michel Riveill, Christian Martel, *Towards Unanticipated Dynamic Service Adaptation,* Third International Workshop on Coordination and Adaptation Techniques for Software Entities (in conjunction with ECOOP'06) (WCAT'06), pp. 25-34, Nantes, France, July 2006.

[44] Clemens Szyperski, *Component Software: Beyond Object-Oriented Programming.* 2nd ed., Addison-Wesley. 2002. ISBN 0-201-74572-0.

[45] OASIS - Advanced Open Standards for the Information Society, http://www.oasis-open.org/home/index.php (accessed on 26.08.2009).

[46] Hanan Lutfiyya, Gary Molenkamp, Michael Katchabaw, and Michael Bauer, *Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework,* In proceedings of the 2nd International Workshop on Policies for Distributed Systems and Networks (POLICY '01), Springer-Verlag. pp. 185 - 201, 2001.

[47] FIP TC-2 Workshop on Architecture Description Languages (WADL), World Computer Congress, Toulouse, France, 2004.

[48] Christos Efstratiou, Adrian Friday, Nigel Davies and Keith Cheverst, *Utilizing the Event Calculus for Policy Driven Adaptation on Mobile Systems,* In proceedings of 3rd International Workshop on Policies for Distributed Systems and Networks, pp. 13–24, 2002.

[49] Lalana Kagal, Tim Finin, Anupam Joshi, *A Policy Language for a Pervasive Computing Environment,* policy, pp.63, Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'03), 2003.

[50] Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin and Amin M. Vahdat, *Model-based Resource Provisioning in a Web Service Utility,* In proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems. 2003.

[51] Abhishek Chandra, Weibo Gong and Prashant Shenoy, *Dynamic Resource Allocation for Shared Data Centers Using Online Measurements,* LNCS Quality of Service - IWQoS 2003, 11th International Workshop Berkeley, CA, USA, Springer Verlag, vol. 2707, pp. 381 – 398, June 2003.

[52] Eithan Ephrati and Jeffrey S. Rosenschein, *Divide and Conquer in Multi-agent Planning,* In proceedings of National Conference on Artificial Intelligence, Seattle, WA, USA pp. 375-380, 1994.

[53] Terence Kelly, *Utility-directed Allocation,* In proceedings of the First Workshop on Algorithms and Architectures for Self-Managing Systems, June 2003.

[54] Brian D. Noble and Mahadev Satyanarayanan, *Experience with Adaptive Mobile Applications in Odyssey,* Mobile Networks and Applications, vol. 4, no. 4, pp. 245-254, 1999.

[55] Frank Eliassen, Richard Staehli, Gordon S. Blair, Jan Ø. Aagedal, *QuA: Building with Reusable QoS-aware Components,* OOPSLA Companion, pp. 154-155, 2004.

[56] Sven Koenig, *Topics for Future Planning Competitions,* In proceedings of the ICAPS-03 Workshop on the Competition: Impact, Organization, Evaluation, Benchmarks, Trento, Italy, 2003.

[57] Mourad Alia, Geir Horn, Frank Eliassen, Mohammad U. Khan, Rolf Fricke and Roland Reichle, *A Component-based Planning Framework for Adaptive Systems*, In proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), Montpellier, France, Oct 30 - Nov 1, 2006.

[58] Mohammad U. Khan, Roland Reichle, and Kurt Geihs, *Architectural Constraints in the Model-Driven Development of Self-Adaptive Applications*, IEEE Distributed Systems Online, vol. 9, no. 7, 2008, art. no. 0807-o7001.

[59] Pyrros Bratskas, Nearchos Paspallis, Konstantinos Kakousis and George A. Papadopoulos, *Applying Utility Functions to Adaptation Planning for Home Automation Applications*, In proceedings of the 17th International Conference on Information Systems Development (ISD2008), Paphos, Cyprus, August 25-27, 2008.

[60] Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC), Deliverable D4.2 *System Design of the MUSIC Architecture*, http://www.ist-music.eu/MUSIC/results/music-deliverables/D4.2System%20DesignoftheMUSICArchitecture (accessed on 26.08.2009)

[61] Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC), MUSIC downloads: http://www.ist-music.eu/MUSIC/developer-zone/downloads/music-downloads

[62] Model Driven Architecture Guide v1.0.1, http://www.omg.org/cgi-bin/doc?omg/03-06-01 (accessed on 26.08.2009)

[63] Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC), Deliverable D6.3 *Modelling Notation for Adaptive Applications in Ubiquitous Computing Environment (refined version)*, Editor – Mohammad U. Khan, http://www.ist-music.eu/MUSIC/results/music-deliverables/techreportreference.2009-07-09.2317047181 (accessed on 26.08.2009)

[64] Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC), Deliverable D6.2 *Software Development Method for Adaptive Applications in Ubiquitous Computing Environments (initial version)*, Editor – Michael Wagner, http://www.ist-music.eu/MUSIC/results/music-deliverables/techreportreference.2008-08-01.3918111699 (accessed on 26.08.2009)

[65] Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC), Deliverable D6.4 *Modelling Notation and Software Development Method for Adaptive Applications in Ubiquitous Computing Environments*, Editor – Michael Wagner, (to appear at http://www.ist-music.eu/MUSIC/results/music-deliverables/)

[66] North American Industry Classification System (NAICS): http://www.census.gov/epcd/www/naics.html

[67] United Nations Standard Products and Services Code (UNSPSC): http://www.unspsc.org

[68] MOFScript (2006). MOFScript Eclipse plug-in homepage, http://www.modelbased.net/mofscript (cited November 2006)

[69] Modelware (2003-2006). MODELling solution for softWARE systems, ESPRIT FP6-IP 511731 project, http://www.modelware-ist.org

[70] Jon Oldevik, Tor Neple, Roy Grønmo, Jan Aagedal and Arne-J. Berre, *Toward Standardised Model to Text Transformations,* In proceedings of the European

Conference on Model Driven Architecture - Foundations and Applications, Nuremberg, ISBN 3-540-30026-0, pp. 239-253, November 2005.

[71] OMG Meta Object Facility (MOF) 2.0 Query/View/Transformation specification (QVT). Final adopted specification, OMG document: http://www.omg.org/docs/ptc/05-11-01.pdf (accessed on 08.09.2009)

[72] OMG Model2Text RFP (2004). MOF Model to Text Transformation Language RFP. OMG document ad/04-04-07, http://www.omg.org/cgi-bin/doc?ad/04-04-07.pdf (accessed on 05.11.2009).

[73] Enterprise Architect, UML Modeling tool from Sparx Systems, http://www.sparxsystems.com

[74] Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments (MUSIC), Deliverable D7.2 *MUSIC Studio and Tools (initial version)*, Editor – Bert Vanhooff.

[75] OSGi – The Dynamic Module System for Java, http://www.osgi.org/Main/HomePage (accessed on 03.11.2009)

[76] Apache Maven, http://maven.apache.org (accessed on 03.11.2009)

[77] Protégé Ontology Editor, http://protege.stanford.edu (accessed on 03.11.2009)

[78] Papyrus UML Modeling tool, http://www.papyrusuml.org (accessed on 03.11.2009)

[79] XMI, http://www.omg.org/spec/XMI/2.1.1 (accessed on 03.11.2009)

[80] The Jena project, http://jena.sourceforge.net (accessed on 03.11.2009)

[81] The Kazuki project, http://projects.semwebcentral.org/projects/kazuki (accessed on 03.11.2009)

[82] RDFReactor, http://semanticweb.org/wiki/RDFReactor (accessed on 03.11.2009)

[83] Jenabean, http://code.google.com/p/jenabean (accessed on 03.11.2009)

[84] PhoneME Java ME Platform, https://phoneme.dev.java.net (accessed on 03.11.2009)

[85] Knopflerfish OSGi Service Platform, http://www.knopflerfish.org (accessed on 03.11.2009)

[86] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional.

[87] MUSIC, MUSIC Development Environment, http://www.ist-music.eu/MUSIC/developer-zone/documentation/downloads/MUSIC%20development%20environment.pdf (accessed on 26.08.2009)

[88] Yves Vanrompay, Manuele Kirsch-Pinheiro, Yolande Berbers, *Context-Aware Service Selection with Uncertain Context Information*, In Electronic Communications of the EASST, Vol. 19, 2009.

[89] Nearchos Paspallis, *Middleware-based Development of Context-aware Applications with Reusable Components*, PhD thesis, University of Cyprus, 2009, http://www.cs.ucy.ac.cy/~paspalli/phd/paspallis_phd_thesis_2009-submitted.pdf (accessed on 03.11.2009)

[90] Romain Rouvoy, Mikaël Beauvois, Frank Eliassen, *Dynamic aspect weaving using a planning-based adaptation middleware*, Proceedings of the 2nd workshop on Middleware-application interaction: affiliated with the DisCoTec federated conferences, 2008, ISBN:978-1-60558-204-7, pp. 31-36, Oslo, Norway.

[91] Javier Cámara, Carlos Canal, Javier Cubo, Juan M. Murillo, *An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution*, Electronic Notes in Theoretical Computer Science (ENTCS), vol. 189 (July 2007), pp. 21-34.

[92] Matti A. Hiltunen and Richard D. Schlichting, *Adaptive Distributed and Fault-tolerant Systems*, International Journal of Computer Systems Science and Engineering, vol. 11, pp. 125–133, September 1996.

[93] Gregor Kiczales and Mira Mezini, *Aspect-Oriented Programming and Modular Reasoning,* In 27th Int. Conf. on Software Engineering (ICSE), pages 49–58. ACM, May 2005.

[94] Kurt Geihs, Paolo Barone, Frank Eliassen, Jacqueline Floch, Rolf Fricke, Eli Gjorven, Svein Hallsteinsen, Geir Horn, Mohammad U. Khan, Alessandro Mamelli, George A. Papadopoulos, Nearchos Paspallis, Roland Reichle, Erlend Stav, *A Comprehensive Solution for Application-Level Adaptation*, Journal on Software Practice and Experience, 2008.

[95] Kurt Geihs, Roland Reichle, Michael Wagner, Mohammad U. Khan, *Modeling of Context-Aware Self-Adaptive Applications in Ubiquitous and Service-Oriented Environments*, In: Software Engineering for Self-Adaptive Systems (SefSAS), ed. by Betty H.C. Cheng, Rogerio de Lemos, Holger Giese, Paola Inverardi, Jeff Magee. Springer-Verlag, LNCS 5525, chap. 8, pp. 146-163, 2009.

[96] Romain Rouvoy, Paolo Barone, Yun Ding, Frank Eliassen, Svein Hallsteinsen, Jorge Lorenzo, Alessandro Mamelli and Ulrich Scholz, *Modeling of Context-Aware Self-Adaptive Applications in Ubiquitous and Service-Oriented Environments*, In: Software Engineering for Self-Adaptive Systems (SefSAS), ed. by Betty H.C. Cheng, Rogerio de Lemos, Holger Giese, Paola Inverardi, Jeff Magee. Springer-Verlag, LNCS 5525, chap. 8, pp. 146-163, 2009.

[97] Dominik Kuropka and Mathias Weske, *Implementing a Semantic Service Provision Platform — Concepts and Experiences,* Wirtschaftsinformatik Journal, Issue 1/2008, pp. 16–24.

[98] Oliver Moser, Florian Rosenberg, and Schahram Dustdar, *Non-intrusive monitoring and service adaptation for WS-BPEL,* In 17th Int. Conf. on World Wide Web (WWW), ACM. 2008.

[99] Steffen Bleul and Thomas Weise, *An Ontology for Quality-Aware Service Discovery,* In First International Workshop on Engineering Service Compositions (WESC'05), IBM Report RC23821, pages 35–42, December 2005.

[100] Daniel A. Menasce and Vinod Dubey, *Utility-based QoS Brokering in Service Oriented Architectures,* icws, pp.422-430, IEEE International Conference on Web Services (ICWS 2007), 2007.

[101] Seyed M. Sadjadi and Philip. K. McKinley, *ACT: An Adaptive CORBA Template to Support Unanticipated Adaptation,* In Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS'04), Tokyo, Japan, March 2004.

[102] Mario Pukall, Christian Kästner and Gunter Saake, *Towards Unanticipated Runtime Adaptation of Java Applications*, Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference, p.85-92, December 03-05, 2008.

[103] Damien Conroy, *The use of domain level semantics to support unanticipated system adaptation,* Conference on Object Oriented Programming Systems Languages and Applications, 2002, pp. 4-5.

[104] Paul Bachmann, *Die analytische Zahlentheorie,* Leipzig: Teubner, 1894.

[105] Roland Reichle, Michael Wagner, Mohammad U. Khan, Kurt Geihs, Massimo Valla, Cristina Fra, Nearchos Paspallis and George A. Papadopoulos, *A Context Query Language for Pervasive Computing Environments*, 5th IEEE Workshop on Context Modeling and Reasoning (CoMoRea) in conjunction with the 6th IEEE International Conference on Pervasive Computing and Communication (PerCom), Hong Kong, 17–21 March 2008, IEEE Computer Society Press, pp. 434-440.

[106] Ulrich Scholz and Romain Rouvoy, *Divide and Conquer – Organizing Component-based Adaptation in Distributed Environments,* In Electronic Communications of the EASST, Vol. 12, 2008.

Part IV Appendices

A Updated Middleware Source Code

In this appendix, we have presented part of the source code that is updated from the MUSIC middleware in order to support the new adaptation reasoning mechanism and the runtime creation of the variability model through installation of application bundles and matching among bundle artifacts.

A.1 Creation of the Variability Model

A.1.1 Installation of a Bundle

```
public void install(URL location) throws MusicException {
    if (location == null)
        throw new MusicException("To install a bundle you need to specify a valid location");
    logger.debug("Installing bundle from: " + location);
    try {
        // Install the bundle
        Bundle bundle = ctxt.getBundleContext().installBundle(location.toString());
        // Start the bundle
        bundle.start();
    } catch (Throwable t) {
        logger.error("Error when installing the bundle from: " + location, t);
        throw new MusicException("Error when installing the bundle from: " + location);
    }
    logger.info("Installed bundle from: " + location);
}
```

A.1.2 Installation of Bundle Artifacts

```
public void installArtifacts(IBundle iBundle) throws MusicException {
    // Installing plans
    IPlan[] plans = iBundle.getPlans();
    if (plans != null) {
        for (int i=0; i<plans.length; i++)
            addIPlan(plans[i]);
    }
    // Installing componentTypes
    ComponentType[] compTypes = iBundle.getComponentTypes();
    if (compTypes != null) {
        for (int i=0; i<compTypes.length; i++)
            addComponentType(compTypes[i]);
    }
    // Installing applications
    ApplicationType[] appTypes = iBundle.getApplicationTypes();
    if (appTypes != null) {
        for (int i=0; i<appTypes.length; i++)
            addApplicationType(appTypes[i]);
    }
    logger.info("The artefacts of the MUSIC bundle have been installed");
}
```

A.1.3 Adding Plans to the Repository

```
protected void addIPlan(IPlan iPlan) {
    ArrayList matchedComponentTypes = matchPlanWithComponentType(iPlan);
    ArrayList matchedApplicationTypes = matchPlanWithApplicationType(iPlan);
    if(matchedComponentTypes.size()>0 || matchedApplicationTypes.size()>0){
        //Update the component type repository
        for(int i=0; i<matchedComponentTypes.size(); i++){
            planRepository.register(new String(((ComponentType)matchedComponentTypes.get(i)).
                getTypeName().toString()), iPlan);
        }
        //Update the application type repository
        for(int i=0; i<matchedApplicationTypes.size(); i++){
            planRepository.register(new String(((ApplicationType)matchedApplicationTypes.get(i)).
                getTypeName().toString()), iPlan);
        }
    } else {
        planRepository.register(new String("NOTMATCHED"), iPlan);
    }
}
```

A.1.4 Matching a Plan with Component Types

```
protected ArrayList matchPlanWithComponentType(IPlan plan){
    ArrayList matchedTypesList = new ArrayList();
    String[] funcsPlan = plan.getFunctionalities();
    String[] propertyTypes = plan.getPropertyTypes();
    Object[] compTypeNames = componentTypeRepository.list();
    for(int i = 0; i<compTypeNames.length; i++) {
        Object[] typesTemp = componentTypeRepository.resolveAll(compTypeNames[i],
            null).toArray();
        ComponentType[] types = new ComponentType[typesTemp.length];
        for(int p=0; p<typesTemp.length; p++){
            types[p] = (ComponentType)typesTemp[p];
        }
        for(int j=0; j< types.length; j++){
            String[] funcsType = types[j].getFunctionalities();
            boolean[] funcCoverage = new boolean[funcsType.length];
            boolean matched = true;
            Arrays.fill(funcCoverage, false);
            for(int k = 0; k<funcsPlan.length; k++){
                for(int l=0; l<funcsType.length; l++){
                    if(funcsPlan[k].equals(funcsType[l]))funcCoverage[l]= true;
                }
            }
            for(int m=0; m<funcCoverage.length; m++){
                if(funcCoverage[m] == false){
                    matched = false;
                }
            }
            if(matched){
                    matchedTypesList.add(types[j]);
            }
        }
    }
    return matchedTypesList;
}
```

A.1.5 Adding an Application Type

```
protected void addApplicationType(ApplicationType applicationType) {
    //addComponentType(applicationType);
    updatePlanRepositoryWithNewTypes(applicationType);
    Map appProperties = new HashMap();
    appProperties.put(IApplicationStatus.APPLICATION_STATUS, new
        Integer(IApplicationStatus.APPLICATION_STOPPED));
    applicationTypeRepository.register(applicationType.getTypeName(), applicationType,
        appProperties);
    if(applicationTypeRepository.list() !=null){
        System.out.println("APPLICATION REGISTERED");
    }
}
```

A.1.6 Updating the Plan Repository

```
protected void updatePlanRepositoryWithNewTypes(ComponentType componentType){
    if(componentType instanceof ApplicationType){
        System.out.println("Updating for application type: "+componentType.getTypeName());
    }
    Object[] planIdentifiers = planRepository.list();
    for(int i = 0; i<planIdentifiers.length; i++) {
        Object[] plansObj = planRepository.resolveAll(planIdentifiers[i], null).toArray();
        IPlan[] plans;
        plans = new IPlan[plansObj.length];
        for (int p=0; p<plansObj.length; p++){
            plans[p] = (IPlan)plansObj[p];
        }
        for(int j=0; j<plans.length; j++){
            String[] funcsPlan = plans[j].getFunctionalities();
            String[] funcsType = componentType.getFunctionalities();
            boolean[] funcCoverage;
            funcCoverage = new boolean[funcsType.length];
            boolean matched = true;
            Arrays.fill(funcCoverage, false);

            for(int k = 0; k<funcsPlan.length; k++){
                for(int l=0; l<funcsType.length; l++){
                    if(funcsPlan[k].equals(funcsType[l]))funcCoverage[l]= true;
                }
            }
            for(int m=0; m<funcCoverage.length; m++){
                if(funcCoverage[m] == false){
                    matched = false;
                }
            }
            if(matched){
                System.out.println("Match found plan update! "+ "Component Type:"+componentType.
                    getTypeName()+" with Plan: "+((IPlan)plans[j]).getName());
                planRepository.register(new String (componentType.getTypeName().toString()),
                    ((IPlan)plans[j]));
            }
        }
    }
}
```

A.2 Adaptation Reasoning

A.2.1 Initiation of Building Templates

```
public synchronized ConfigurationTemplate buildTemplates(MusicName type,
        AdaptationResourceDescriptor[] descriptors, Map filters, IContextValueAccess context) {
    for(int i = 0; i<1; i++){
        addApplicationType(type);
    }
    // Store node addresses of resources in a SetMap according to node types
    final SetMap resources = new SetMap();
    for (int i = 0; i < descriptors.length; i++)
        resources.getSet(descriptors[i].getNodeType()).add(
                getNodeAddress(descriptors[i]));
    for (final Iterator it = resources.keySet().iterator(); it.hasNext();) {
        Object key = it.next();
        Set v = resources.getSet(key);
        nodesMap.put(key, v.toArray(new String[v.size()]));
    }
    // Ensure that the root component is deployed on the master node
    String[] local = (String[]) nodesMap.get(ResourceVocabulary.MASTER_NODE_TYPE);
    ConfigurationTemplate conf = getBestTemplate(type, descriptors, filters, context, local);
    return conf;
}
```

A.2.2 Retrieval of the Best Template

The getBestTemplate() method

```
protected ConfigurationTemplate getBestTemplate(MusicName type,
        AdaptationResourceDescriptor[] descriptors, Map filters, IContextValueAccess context,
            String[] nodes){
    ConfigurationTemplate bestTemplate = null;
    HashMap bestTemplateMap = getBestTemplateWithUtility(type, descriptors, filters, context,
        nodes);
    if((bestTemplateMap == null)|| (bestTemplateMap.keySet().size() == 0)){
        return null;
    }
    Set keys = bestTemplateMap.keySet();
    Object[] keysArray = keys.toArray();
    for(int i=0; i<keysArray.length; i++){
        bestTemplate = (ConfigurationTemplate)bestTemplateMap.get(keysArray[i]);
    }
    return bestTemplate;
}
```

The getBestTemplateWithUtility() method

```
protected HashMap getBestTemplateWithUtility(MusicName type,
        AdaptationResourceDescriptor[] descriptors, Map filters, IContextValueAccess context,
            String[] nodes){
    HashMap bestTemplate = null;
    double bestUtility = 0.0;
    double curUtility = 0.0;
    final Set allPlans = getComponentPlans(type);
    for (final Iterator it = allPlans.iterator(); it.hasNext();) {
        final IPlan plan = (IPlan) it.next();
        HashMap templateWithUtility = getTemplateForPlan(plan, descriptors, context, nodes);
        if((templateWithUtility == null)|| (templateWithUtility.keySet().size() == 0)){
            return null;
        }
        Set keys = templateWithUtility.keySet();
        Object[] keysArray = keys.toArray();
        for(int i=0; i<keysArray.length; i++){
            Object key = keysArray[i];
            curUtility = ((Double)key).doubleValue();
            if(curUtility >= bestUtility){
                bestTemplate = templateWithUtility;
                bestUtility = curUtility;
            }
        }
    }
    return bestTemplate;
}
```

A.2.3 Retrieval of the Best Template for Each Plan

```
protected HashMap getTemplateForPlan(IPlan plan, AdaptationResourceDescriptor[] descriptors,
        IContextValueAccess context, String[] nodes){
    HashMap bestTemplateWithUtility = new HashMap();
    IPlanVariant bestVariant = null;
    ConfigurationTemplate bestTemplate = null;
    double bestUtility = 0.0;
    double curUtility = 0.0;
    int nodeIndex = 0;
    Iterator planVariants = plan.planVariants();
    Map childTemplates = new HashMap();
    if (plan instanceof CompositionPlan) {
        while (planVariants.hasNext()){
            Map roleUtilities = new HashMap();
            IPlanVariant curVariant;
            curVariant = (IPlanVariant)(planVariants.next());
            Role[] roles = curVariant.getPlan().getCompositionSpec().getRoles();
            for (int i=0; i < roles.length; i++){
                Map templateAndUtility = getBestTemplateWithUtility(
                        roles[i].getComponentType(), descriptors, null, context, nodes);
                Set keys = templateAndUtility.keySet();
                for(Iterator itt = keys.iterator(); itt.hasNext();){
                    Object key = itt.next();
                    roleUtilities.put(roles[i], key);
                    childTemplates.put(roles[i].getName(), ConfigurationTemplate)templateAndUtility.
                            get(key));
                }
                childTemplates.put(roles[i].getName(), getBestTemplate(
                        roles[i].getComponentType(), descriptors, null, context, nodes));
                    }
                    childTemplates.put(roles[i].getName(), getBestTemplate(
                            roles[i].getComponentType(), descriptors, null, context, nodes));
                }
                ConfigurationTemplate template = new ConfigurationTemplate(curVariant,
                    childTemplates);
                    HashMap weightMap = (HashMap)template.evaluate(IPropertyEvaluator.
                            UTILITY_PROPERTY, context);
                for (int i = 0; i<roles.length; i++){
                    curUtility += (((Double)weightMap.get(roles[i])).doubleValue())*
                    ((Double)roleUtilities.get(roles[i])).doubleValue();
                }
                if(curUtility > bestUtility){
                    bestUtility = curUtility;
                    bestVariant = curVariant;
                }
                if(bestVariant != null){
                    bestTemplate = template;
                }
            }
        } else {
            while (planVariants.hasNext()){
                IPlanVariant curVariant;
                curVariant = (IPlanVariant)(planVariants.next());
                ConfigurationTemplate curTemplate = new ConfigurationTemplate(curVariant);
                Object utilityObj = (curTemplate).evaluate(IPropertyEvaluator.UTILITY_PROPERTY,
                    context);
                Double utilityDbl = (Double)utilityObj;
                curUtility = utilityDbl.doubleValue();
                if(curUtility >= bestUtility){
                    bestUtility = curUtility;
                    bestVariant = curVariant;
                }
            }
            if(bestVariant != null){
                bestTemplate = new ConfigurationTemplate(bestVariant);
            }
        }
        if (bestTemplate != null){
            bestTemplate.setNodeAddress(nodes[nodeIndex]);
            bestTemplateWithUtility.put(new Double(bestUtility), bestTemplate);
        }
        return bestTemplateWithUtility;
}
```

B Publications

In connection with this thesis, I would like to mention the following publications:

1. M. U. Khan, R. Reichle, M. Wagner, K. Geihs, U. Scholz, C. Kakousis and G. A. Papadopoulos, *An Adaptation Reasoning Approach for Large Scale Component-based Applications,* In Electronic Communications of the EASST, Vol. 19, 2009.
2. K. Geihs, R. Reichle, M. Wagner, M. U. Khan, *Modeling of Context-Aware Self-Adaptive Applications in Ubiquitous and Service-Oriented Environments*, In: Software Engineering for Self-Adaptive Systems (SefSAS), ed. by Betty H.C. Cheng, Rogerio de Lemos, Holger Giese, Paola Inverardi, Jeff Magee. Springer-Verlag, LNCS 5525, chap. 8, pp. 146-163, 2009.
3. M. U. Khan, R. Reichle, and K. Geihs, *Architectural Constraints in the Model-Driven Development of Self-Adaptive Applications*, IEEE Distributed Systems Online, vol. 9, no. 7, 2008, art. no. 0807-o7001
4. K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, E. Stav, *A Comprehensive Solution for Application-Level Adaptation*, Journal on Software Practice and Experience, Wiley, 2008.
5. R. Reichle, M. Wagner, M. U. Khan, K. Geihs, J. Lorenzo, M. Valla, C. Fra, N. Paspallis, G. A. Papadopoulos, *A Comprehensive Context Modeling Framework for Pervasive Computing Systems*, 8th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Oslo, Norway, Springer Verlag, June2008
6. R. Reichle, M. U. Khan and K Geihs, *How to Combine Parameter and Compositional Adaptation in the Modeling of Self-Adaptive Applications*, PIK Special Issue: Modeling of Self-Organizing Systems, March, 2008
7. T. Weise, M. Zapf, M. U. Khan, and K. Geihs, *Genetic Programming Meets Model-driven Development*, In Proceedings of Seventh International Conference on Hybrid Intelligent Systems, 17-19 September 2007, Kaiserslautern, Germany
8. M. U. Khan, R. Reichle and K. Geihs, *Applying Architectural Constraints in the Modeling of Self-adaptive Component-based Applications*, Workshop on Model Driven Software Adaptation (M-ADAPT'07) within the 21st European Conference on Object Oriented Programming (ecoop), July 30 - August 03, 2007, Berlin, Germany
9. M. Alia, G. Horn, F. Eliassen, M. U. Khan, R. Fricke and R. Reichle, *A Component-based Planning Framework for Adaptive Systems*, The 8th International Symposium on Distributed Objects and Applications (DOA), Oct 30 - Nov 1, 2006, Montpellier, France
10. K. Geihs, M. U. Khan, R. Reichle, A. Solberg and S. Hallsteinsen, *Modeling of Component-based Self-Adapting Context-Aware Applications for Mobile Devices*, IFIP Working Conference on Software Engineering Techniques, October 17-20, 2006, Warsaw, Poland
11. K. Geihs, R. Reichle, M. U. Khan, A. Solberg and S. Hallsteinsen, *Model-Driven Development of Self-Adaptive Applications for Mobile Devices* (Research Summary), ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), May 21-22, 2006, Shanghai, China
12. K. Geihs, M. U. Khan, R. Reichle, A. Solberg, S. Hallsteinsen and S. Merral, *Modeling of Component-based Adaptive Distributed Applications*, Dependable and Adaptive Distributed Systems (DADS Track) of the 21st ACM Symposium on Applied Computing, April 23 -27, 2006, Bourgogne University, Dijon, France
13. M. U. Khan, K. Geihs, F. Gutbrodt, P. Göhner and R. Trauter, *Model-Driven Development of Real-Time Systems with UML 2.0 and C*, Proceedings of the Fourth Workshop on Model-Based Development of Computer-Based Systems and Third International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES'06) - Volume 00, Pages: 33 - 42, March 30, 2006, Potsdam, Germany.