A Development Framework and Methodology for Self-Adapting Applications in Ubiquitous Computing Environments

S. HALLSTEINSEN[1], K. GEIHS[2], N. PASPALLIS[3], F. ELIASSEN[4], G. HORN[1], J. LORENZO[5], A. MAMELLI[6], G.A. PAPADOPOULOS[3]

[1] SINTEF ICT, Strindveien 4, 7034 Trondheim, Norway – svein.hallsteinsen@sintef.no, geir.horn@sintef.no
[2] University of Kassel, Mönchebergstr. 19, 34109 Kassel, Germany – geihs@uni-kassel.de
[3] University of Cyprus, 1 Panepistimiou Avenue, 2109 Nicosia, Cyprus – nearchos@cs.ucy.ac.cy, george@cs.ucy.ac.cy
[4] University of Oslo, Problemveien 5-7, 0316 Oslo, Norway – frank@ifi.uio.no
[5] Telefónica I+D, Calle Emilio Vargas 6, 28043 Madrid, Spain – jorgelg@tid.es
[6] Hewlett-Packard Italiana Srl, 20063 Cernusco sul Naviglio, Italy – alessandro.mamelli@hp.com

Abstract

Today software is the main enabler of many of the appliances and devices omnipresent in our daily life and important for our well being and work satisfaction. It is expected that the software works as intended, and that the software always and everywhere provides us with the best possible utility. This paper discusses the motivation, technical approach, and innovative results of the MUSIC project. MUSIC provides a comprehensive software development framework for applications that operate in ubiquitous and dynamic computing environments and adapt to context changes. Context is understood as any information about the user needs and operating environment which vary dynamically and have an impact on design choices. MUSIC supports several adaptation mechanisms and offers a model-driven application development approach supported by a sophisticated middleware that facilitates the dynamic and automatic adaptation of applications and services based on a clear separation of business logic, context awareness and adaptation concerns. The main contribution of this paper is a holistic, coherent presentation of the motivation, design, implementation, and evaluation of the MUSIC development framework and methodology.

Keywords

Adaptive Software, Ubiquitous Computing, Model-driven development, Middleware, Mobile computing

1 INTRODUCTION

Software is today the main enabler of many of the appliances and devices omnipresent in our daily life and important for our well being and work satisfaction (Weiser, 1991). Software controls our cars, planes, homes, businesses, and our mobile phones and extensively portable computers (Charette, 2009; ITU, 2009). It is imperative to expect that the software works as intended, and that the software provides us with the largest possible utility always and everywhere, also when users move and the computing and communication environment varies dynamically. The software development practices of today mostly assume a more static view of the user needs and

operating conditions. In ubiquitous computing they must evolve to encompass adaptive applications made for intrinsic user and application mobility as well as context changes at execution time.

This paper discusses the motivation, technical approach, and results of the MUSIC project[1]. The main goal of MUSIC was to simplify the development of adaptive applications that will operate in open and dynamic ubiquitous computing environments and adapt seamlessly and without user intervention in reaction to context changes. The main innovations of MUSIC are a comprehensive development framework that consists of a model-driven development methodology (including a tool chain) for self-adaptive context-aware applications as well as a corresponding extensible context management and adaptation middleware supporting the model-driven development.

Context is understood in a broad sense as any information about the user needs and operating environment which may vary dynamically and impact the applications, and which can be monitored using relevant hardware and software mechanisms. The MUSIC solution supports a variety of adaptation mechanisms, such as setting configuration and application parameters, replacing components and service bindings, and redeploying components on the distributed computing infrastructure.

MUSIC was not the first project addressing self-adaptation of context-aware applications and most of the conceptual and algorithmic groundings were available already. Thus, a substantial part of the MUSIC results are derived from engineering research that has tackled a well-defined problem and has carefully selected and combined existing solutions into a comprehensive development framework for self-adaptive systems. From a general perspective, we claim that MUSIC has produced at least three key achievements that advance the state of the art in software engineering of adaptive software

- clear separation of business logic, context awareness and adaptation concerns,
- automated model-driven development methodology facilitating the design and implementation of context-aware adaptive applications and enabling the re-use of modelling artefacts and adaptive components;
- seamless integration of dynamically discovered external services and their quality of service properties into the adaptation management of component-based applications at run-time,

MUSIC is a successor of the earlier project MADAM[2]. While MADAM focused on mobile computing scenarios with an underlying assumption of a closed world computing environment, MUSIC targets ubiquitous computing environments that are characterized by inherent openness, heterogeneity, and dynamic service discovery and binding. MUSIC reuses many of the concepts and fundamental architectural decisions of MADAM, but refines and extends the framework with a range of new concepts and mechanisms, such as a substantially enhanced context management middleware facilitating the plug-in of new context sensors and reasoners, dynamic service discovery and binding of external services as replacements for local components, and adaptation planning seamlessly taking into account both local components and discovered services and in a coordinated way driven by service level negotiations and non-functional properties of local components.

The main contributions of this paper are a holistic, coherent presentation of the MUSIC development methodology and framework, as well as its evaluation based on trial development and testing of a collection of context-aware, and self-adaptive applications, as researched and developed by the MUSIC project.

The rest of this paper is organized as follows: The next section introduces a motivating scenario and presents the key challenges addressed by the MUSIC project. Then the overall technical approach, system architecture and model-based software development methodology for facilitating the creation and operation of self-adaptive applications are explained. After that we present the implementation of the tools and middleware environment

---

[1] Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments, supported by the European Union under research grant IST-035166 lasting from October 2006 through March 2010.

[2] Mobility and Adaptation-enabling Middleware, supported by the European Union under research grant 004159 lasting from September 2004 through March 2007.

provided by the project, followed by an evaluation of the framework based on the development and experimentation with a set of trial applications. Finally we discuss related work and present conclusions and an outlook to the future of frameworks for self-adaptive applications such as MUSIC.

2 MOTIVATION AND CHALLENGES

As a further illustration of our understanding of the characteristics of context-aware and adaptive software, let us consider a scenario related to travellers on public transportation in urban areas. It is staged in the metro system of a large city and involves passengers using their mobile devices for assistance and entertainment while traveling. Three applications are used in the scenario:

- *TravelAssistant* (*TA*) assists with itinerary planning and ticket purchase, detects travelling delays and notifies the passenger if she or he is affected by such delays. It makes use of external services for itinerary planning, traffic information and maps.
- *InstantSocial* (*IS*) appears as a web site supporting the sharing of photos with other users of the application (Fraga et al., 2008). However, instead of relying on a central Internet server, it is served by a composition of services scattered across nearby devices. As more users participate, this platform becomes more robust, the number of shared content items increases and it becomes more attractive for the users. Each participating device instantiates a smaller or larger set of services depending on its own resource situation and what is available from other devices.
- *ChessMate* is a chess playing program. It may delegate the resource demanding reasoning component to another computer in order to reduce the load on the handheld device.

The networked infrastructure of the train provides connectivity between passenger devices and offers computing resources which can be used by passenger devices to extend their computing capacity. Kiosk computers which can be used by the public – e.g. travellers – are placed in the station by the Metro company. In addition the metro company operates map and itinerary planning services which are used by TravelAssistant.

Figure 1 shows a snapshot of the situation when a train is in the station. The picture focuses on travellers Marie, Paul and Alice. Marie and Paul are on board the train. They are both using InstantSocial to share photos from a concert they both visited the previous night with interested fellow passengers. Marie used TravelAssistant to plan the trip on her way to the station. Since it was bright sunshine outside, the display was unreadable and the device first selected the audio user interface, but switched to the GUI when inside the station. Now it is running in the background monitoring progress and listening for traffic information so it can alert and guide her when she has to change trains and if any event occurs which might cause deviation from the planned itinerary.

Marie's mobile uses the train computer, as an extension of its resources. When she launches InstantSocial, the full configuration is chosen, with the resource demanding content repository (CR) component deployed on the train computer.

The InstantSocial instance on Paul's mobile is a thin client configuration, using the content access (ca) service provided by Marie's instance to access the shared content. This was the initial choice of the middleware, because when he started InstantSocial there were already several instances with the full configuration offering the content access service, so adding another one would not increase the availability of the application significantly. Marie's instance was chosen because due to the assistance from the train computer, the resource utilisation on her device is low.

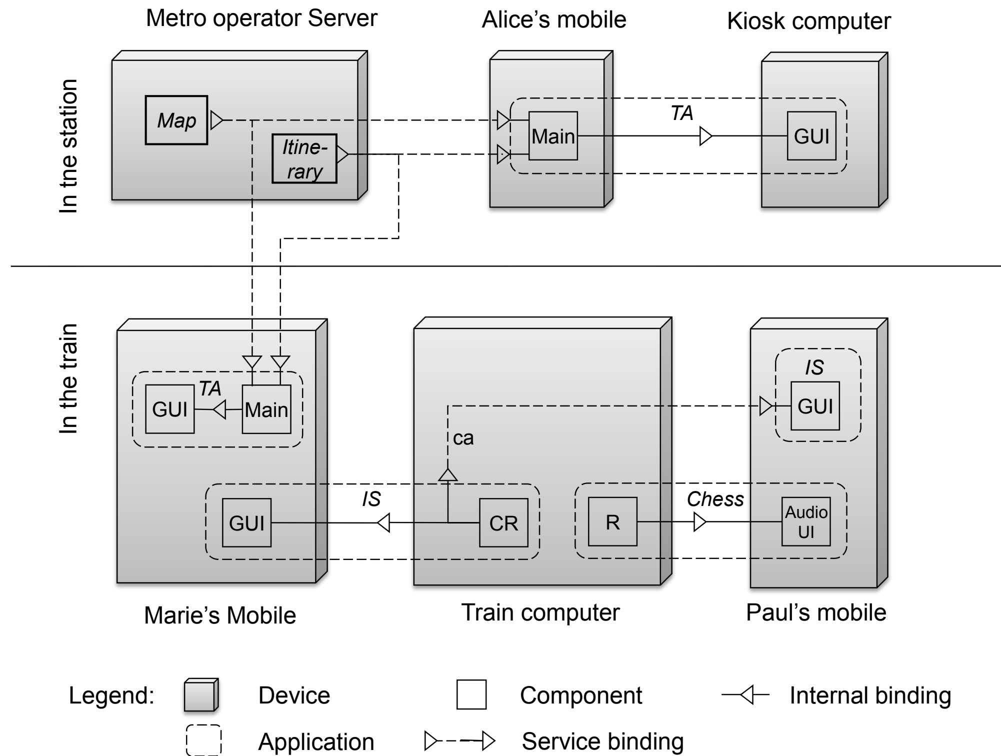


Figure 1: Situation from the scenario

Paul is also using ChessMate to exercise his blind chess skills in parallel with looking at photos. He started the game while waiting for his train at the platform, with the chess playing component deployed on his mobile device. The skill level available there is lower than what Paul prefers, so when the train arrived he decided to start InstantSocial to see if there were any interesting photo sharing communities on the train. On board the train, his device discovered the train computer, where a smarter chess playing component is available. This caused a reconfiguration of the application such that now it is using the reasoning component on the train computer instead. As a result, the skill level of Paul's opponent has increased to the preferred level.

On the platform, Alice is planning her journey using TravelAssistant. She started the application on her mobile device while descending to the station on the escalator, but when she passed a kiosk computer on the platform, she accepted the suggestion from the application to take advantage of its larger and easier to use display instead, and her device adapted accordingly. When she walks away from the kiosk the user interface on the kiosk is stopped and the user interface returns to the mobile. When done with the itinerary, Alice starts her media player to listen to music while waiting for her train to arrive. The TravelAssistant is reconfigured again and switches to the background mode with the itinerary planned at the kiosk, listening to traffic information. After a few stations, the device discovers that there is an incident on the metro line planned for the next leg of Alice's itinerary causing a delay, and that there is an alternative itinerary which will take her to her destination station earlier. Alice is alerted about this and accepts the new itinerary.

This means that she has to change trains at the next station, and she uses TravelAssistant to navigate through the station to the platform for the next train. Now TravelAssistant needs much more resources, and since it is now in the foreground, the audio player is reconfigured to a pause configuration, keeping its state but using a minimum of resources.

Developing applications that behave as illustrated above is considerably more complicated than developing normal non-adapting applications. Reasons are:

- The addition of self-adaptation capabilities to a system adds considerably to the complexity. In many cases this may be even more challenging than developing the functionality of the system itself.
- The business logic of applications must be designed in a way that allows adaptation. There is a range of different adaptation mechanisms that applications may want to exploit, such as adapting a parameter value, the behaviour, the deployment, the service selection, etc. This choice of adaptation techniques adds to the complexity.
- Self-adaptive systems may be large and long lived, so established engineering practices based on modularity, component reuse and evolution should not be compromised.
- The applications must monitor the relevant context so that they are aware of the changes that impact the adaptation.
- In most situations where adaptation is desirable, there are several factors influencing the decision, and the balancing of the different factors to make a good decision requires complex computations.
- When several applications are running concurrently in a device, they are competing for the available resources, and thus their adaptation should be coordinated.
- The additional computational burden of adaptation may cause problems related to e.g. performance, scalability, robustness, usability, etc.

In conclusion, since none of the existing solutions for application adaptivity provided all of the desired features and no development methodology was available that specifically targeted adaptive applications in ubiquitous computing environments, the need for a new comprehensive and coherent development and runtime support for self-adaptive applications became obvious.

3 THE MUSIC SOLUTIONS

To meet the challenges identified in the previous section, MUSIC has developed a comprehensive *software development framework that* enables automation of the adaptation of the software to the varying user needs and operating conditions at runtime. An overview of the MUSIC framework is given in Figure 2. The development of adaptive applications is alleviated by extensive support from the framework, including

- a modelling language which supports separation of concerns; and where self-adaptation and business logic are addressed separately to avoid the surge in complexity;
- generic, reusable middleware components which automate context monitoring and management, and adaptation;
- tools which support the development of design models annotated with context and adaptation concerns and transform them into run-time knowledge available to the middleware.

The middleware implements a control loop which complies with the well-known MAPE (Monitor, Analyse, Plan, Execute) loop in autonomic computing (Kephart and Chess, 2003). It monitors the relevant context sensors, and when significant changes are detected, it triggers a planning process to decide if adaptation is necessary. When this is the case the planning process finds a new configuration that fits the current context better than the one that is currently running, and triggers the adaptation of the running application. To do this the middleware relies on an annotated QoS aware architecture model of the application available at runtime, which specifies its adaptation capabilities and its dependencies on context information. This model corresponds to the “Knowledge” component of the autonomic manager in the autonomic computing blueprint. The planning process evaluates the utility of alternative configurations, selects the most suitable one for the current context (i.e. the one with the highest utility for the current context which does not violate any resource constraints) and adapts the application accordingly.

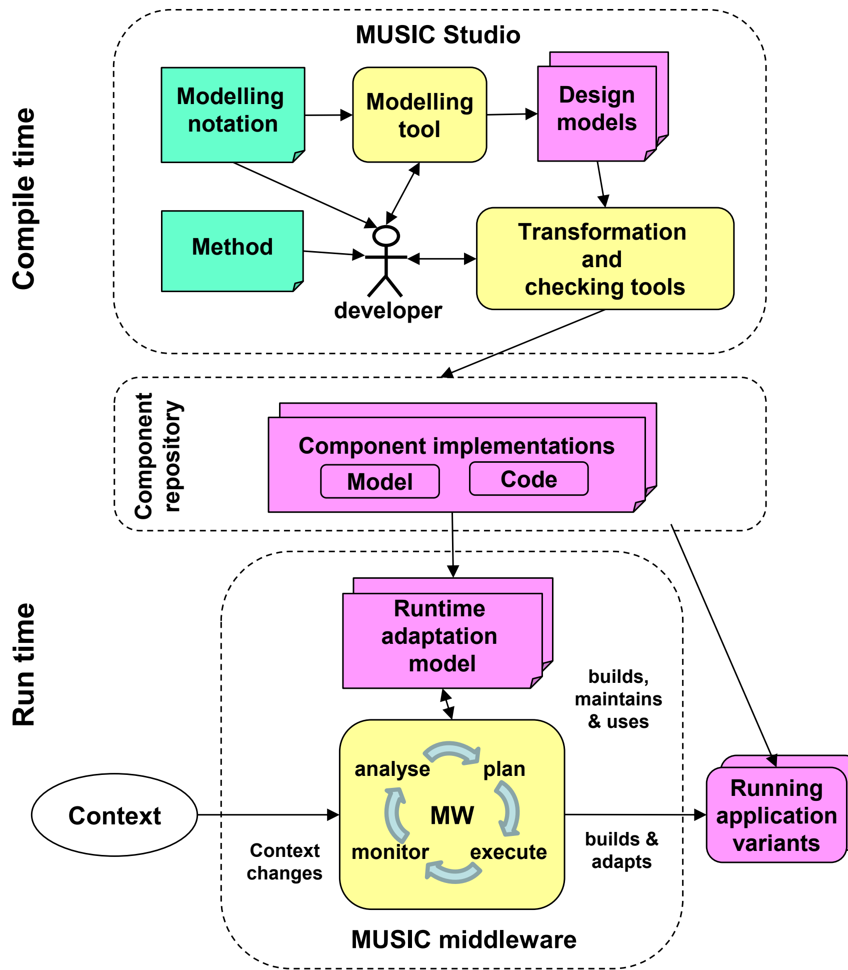


Figure 2: Overview of the MUSIC development framework

One of the unique features of MUSIC is that a number of different adaptation mechanisms are supported, all included in a single adaptation framework, including parameter setting, component replacement and redeployment (Floch, 2006; Geihs et al., 2009a), and service rebinding (Rouvoy et al., 2008).

3.1 Adaptation model

The *QoS-aware adaptation model* describes the relevant QoS dimensions and how they are affected when varying the actual configuration and the context. The selection of the most suitable configuration is based on the evaluation of a utility function which measures how well suited a configuration is for a given context (Geihs et al., 2009a). In this section we describe the model, and show how it is used by the middleware, using excerpts of the models of the TravelAssistant and InstantSocial applications as examples. The notation used in the examples is explained in Figure 3.

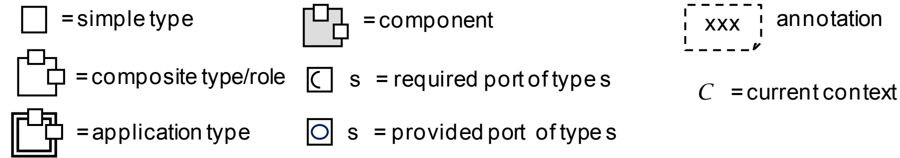


Figure 3: Notation used in the examples

An application is modelled as a component framework, which describes an abstract composition of functionalities expressed as a set of typed *Roles* collaborating through *Ports*. The latter represent either functionality provided to or required from collaborating roles. The typed roles can be dynamically configured with conforming role realizations (i.e. implementations) and deployed on different nodes at runtime. A port has a *Type* defining the functionality represented by the port in terms of interfaces and protocol. Components implement ports and a component can be used in a role if the ports match (same type). For example, in the model for the TravelAssistant application depicted in Figure 4 there are two roles typed UI and Ctrl. The UI role, representing the user interface, has one port providing the UI functionality. The Ctrl role has a port requiring the UI functionality which is connected to the providing port of the UI role.

Applications may be distributed on several devices. In this case a *master* node is in charge of the adaptation reasoning and instructs the adaptation of the other nodes, called *slaves*. The master node is usually a handheld device carried by the user. Master nodes discover slave nodes dynamically and include them in their *adaptation domain* if the master runs applications which may make use of the slave. Slave nodes run a special version of the middleware. In the TravelAssistant model the deployment annotation of the UI role specifies that this component may be deployed either on the handheld or on a slave node. In the scenario, the kiosk computer is set up as a slave, and when Alice approaches it, her smartphone, which is set up as a master, discovers it and includes it in its adaptation domain. This enables the configuration with the user interface deployed on the slave.

Properties and property predictor functions associated with the ports specify how to compute the QoS properties and resource needs of components and compositions. The predictor functions are expressed as expressions over the properties of the context and components in collaborating roles, and in the case of composite components, also the properties of the components in their inner roles.

The relevant properties for the TravelAssistant model depicted in Figure 4 are declared at the top of the figure. The UI type is annotated with the properties distinguishing the alternative implementations of this type. It has three alternative implementations, nmlGUI which is the normal graphical UI for the smartphone touchscreen, AudioUI which is for outdoor use when light conditions makes it difficult to read the smartphone screen, and BigUI which is suited for a larger screen. The associated property predictors reflect this. The question mark in the property predictor for nmlGUI and audioUI indicates that this property is configurable by the middleware.

The meaning – and significance - of a context change is defined by the developer, e.g., the developer can deem some context changes as less significant in the sense that they will not trigger the planning process, while others will. The developer specifies the context dependencies indirectly in the property predictors and the utility functions; see Section 4.1 for details. In practice, the context-aware behaviour of an application is expressed by means of properties that depend on available context parameters. These dependencies are defined by the application developer as part of the adaptation model. In the TravelAssistant example for instance the light and the vibration properties are used to characterise both the software and the context. As a context property, vibration expresses a user preference available from the user profile, while as a software property it expresses whether the software uses vibration to alert the user or not.

	name	Repr.	Context	description
Properties	audio	bool		Using spoken dialog?
	light	float	Ambient light	Light strength in the environment
	vibration	bool	User profile	Using vibration to alert the user?
	cost	float		Cost of use
Resources	screen	{small, big }		Size of the screen

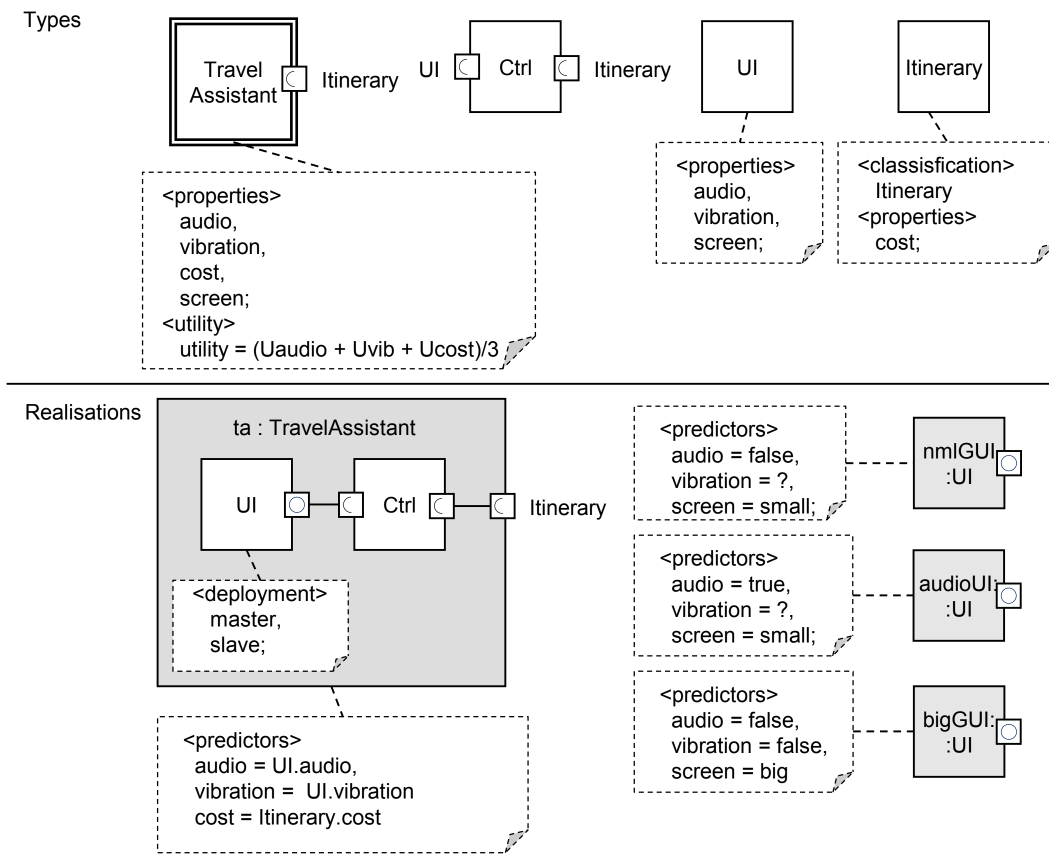
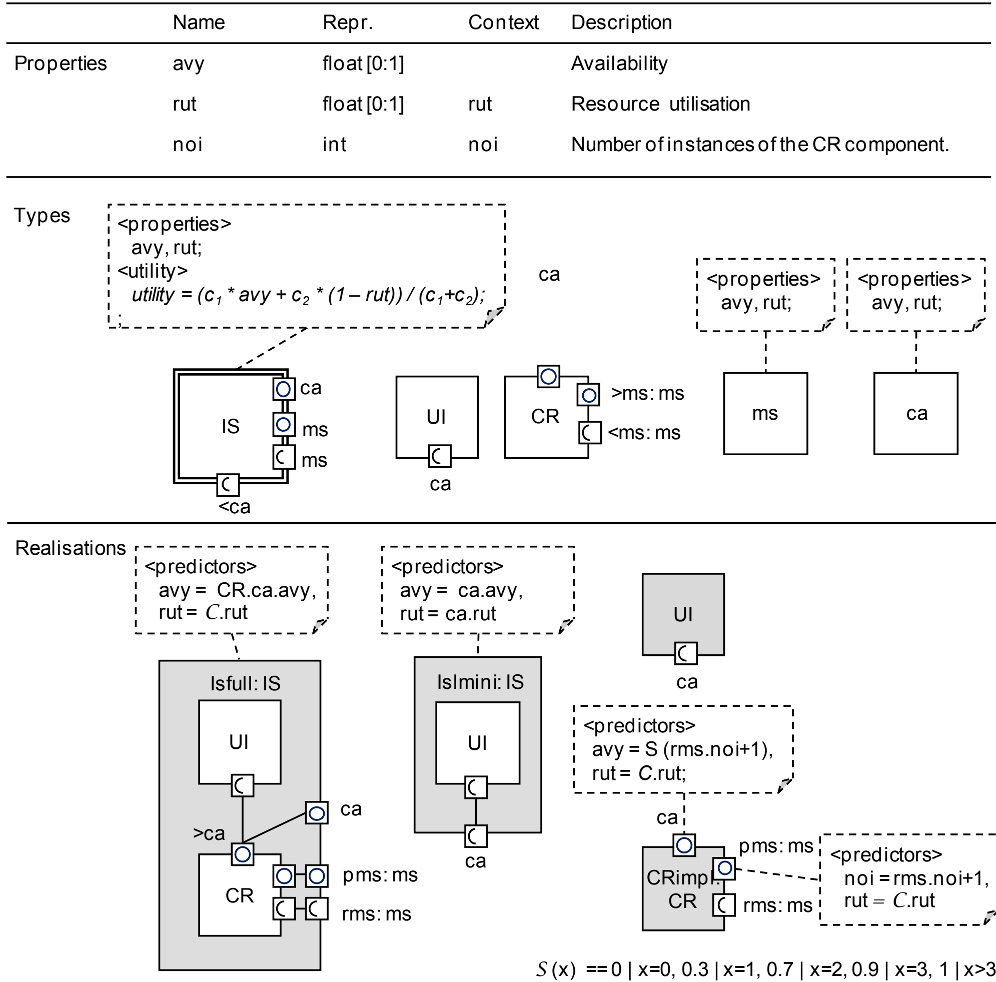


Figure 4: Partial annotated architecture model for the TravelAssistant application

The utility function compares the properties of a configuration of the application with the properties of the current context and computes a utility value expressing how well suited the configuration is in the current context. The utility function is provided by the developer and is typically expressed as a weighted sum of dimensional utility functions where the weights express user preferences (i.e., relative importance of a dimension to the user). A dimensional utility function measures user satisfaction in one property dimension. For example, the utility function of the TravelAssistant application sums the utility of the audio, vibration and cost properties using the same weight for all properties.

A prominent feature of MUSIC is its combination of component-based software engineering with service-oriented architectures (SOA) to allow applications on mobile devices to adapt to and benefit from discoverable services in their proximity. Note that these services may be provided by other MUSIC nodes as well as - via standard protocols - by nodes that do not run the MUSIC middleware (see Section 3.3.3). Ports at the border of an



	Name	Repr.	Context	Description
Properties	avy	float [0:1]		Availability
	rut	float [0:1]	rut	Resource utilisation
	noi	int	noi	Number of instances of the CR component.

Figure 5: Partial annotated architecture model for the InstantSocial application

application represent services provided or required by the application. Services are described by types, and their QoS levels are described by properties, in the same way as with components.

TravelAssistant uses an itinerary planning service provided by the metro operating company. As illustrated in Figure 4, this is modelled by the port typed Itinerary at the application border. There may be different versions of this service available at runtime, differing in cost of use as indicated by the property annotation of the service type. The middleware takes care of the *discovery* and *publication* of services and notifies clients about changes in properties. The classification annotation associated with the service type instructs the middleware about how to discover services of this type.

The InstantSocial application from the scenario provides a more sophisticated example of the use of services. Different instances of this application provide services to and use services from each other to create a shared photo gallery distributed on the participating devices while seeking to distribute the computational load in accordance with the resource situation on the devices. A partial model for InstantSocial is depicted in Figure 5.

The model defines two component types, UI (user interface) and CR (content repository). The content repository component is responsible for maintaining and providing access to an inventory of available content in all the participating devices and for providing access to it. CR instances act both as consumers and providers of the membership (ms) service. When a new CR instance is created, it will use the membership service provided by an existing instance to become included in the common distributed content repository, and later possibly provide this

service to another new instance. CR instances also implement partial replication of content to ensure a certain stability of the federated repository even when participants leave. There are two alternative realisations for the application, labelled IsFull and IsMini. IsFull includes both the UI and the CR components, and thus contributes to the implementation of the shared photo gallery, while IsMini includes only the UI component and relies on using the content access (ca) service provided by another IS instance.

This example demonstrates the capability of a group of MUSIC master nodes to do coordinated adaptations of a service based system distributed over the corresponding adaptation domains. The coordination is facilitated by the QoS aware service publication and discovery mechanism. When the middleware publishes a service it includes the properties computed by the property predictors of the providing application. If the properties change, as a result of a context change or/and a reconfiguration on the provider side, the clients are notified. When MUSIC applications use services provided by non-MUSIC service providers, it is assumed they behave in the same way. In the example, the property annotations and the utility function are designed to balance the burden of providing the shared photo gallery in accordance with the resource situation on the participating devices.

Component realizations are either *Atomic* or *Composite*. A *Composite Realization* is itself an abstract composition and allows for recursive decomposition. Often there are dependencies between the choices of realisations in different roles in a composition. This can be expressed by *Constraints,* which are predicates over the properties of the constituting components of a composition, in effect restricting the possible combinations of component realizations (Geihs et al., 2009a).

The user of a master node may start (instantiate) and stop (remove) MUSIC applications, causing the set of running applications inside the adaptation domain to change and be adapted by the middleware in accordance with these user actions, relevant context changes, and resource constraints, seeking to maximise the overall utility at any time. Adaptation planning will be triggered during the launch of the application and at run-time when the middleware detects a significant change in the execution context. In this case the middleware evaluates the suitability of all feasible configurations in the current execution context by computing the *Predicted Properties and the* expected utility of the evaluated application configuration (Brataas et al., 2007; Geihs et al., 2009a), and reconfigures the applications accordingly. The overall utility of the set of running applications is computed as a weighted sum of the utilities of the running applications, where the weights reflect the relative priorities of the applications. These priorities are controlled by the user through the user profile.

In a ubiquitous and service oriented computing environment, there will typically be churn of devices and service providers causing adaptations. If a slave node or a service provider fails during an application reconfiguration, the completion of the planned reconfiguration might become impossible. In this case a new adaptation is triggered to overcome this temporary situation. Integrity concerns of application state due to loss of application components hosted by the failing slave node must be handled by the application itself. If a master node fails, the corresponding applications are lost. Slave nodes will eventually learn about the failure and discard running components they are hosting for the failed master node.

Serving external clients also consumes resources in the adaptation domain. Therefore whether or not to publish a service outside the adaptation domain is also decided at runtime by the adaptation middleware. For example, a node running InstantSocial may select the ISfull configuration to ensure availability, despite being short on resources, while deciding not to publish its membership service to avoid the burden of serving as an entry point for new entrants[3].

3.2 Model-driven development methodology

MUSIC was the first project that delivered a development methodology facilitating specifically the development of adaptive applications in open, heterogeneous ubiquitous computing environments. In order to

[3] This is not included in the example model in Figure 5.

support application developers during the creation of the application adaptation model and the deployment on the middleware, the MUSIC solution provides methodology support in accordance with the model-driven software development paradigm. Using a specific UML-based modelling notation, the application developer is enabled to specify the application variability model, context elements and data structures, as well as service functionalities and QoS-properties at an abstract and platform-independent level. The necessary source code to publish the application's adaptation capabilities, context dependencies, and variability features with regard to external services to the 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 of the adaptations. The modelling support and code generation facilities provided by MUSIC focus on the adaptation capabilities of an application. A general model-driven development approach for the functional parts of the applications is well covered by the current state of the art in software engineering. Therefore, it is not addressed specifically by the MUSIC methodology.

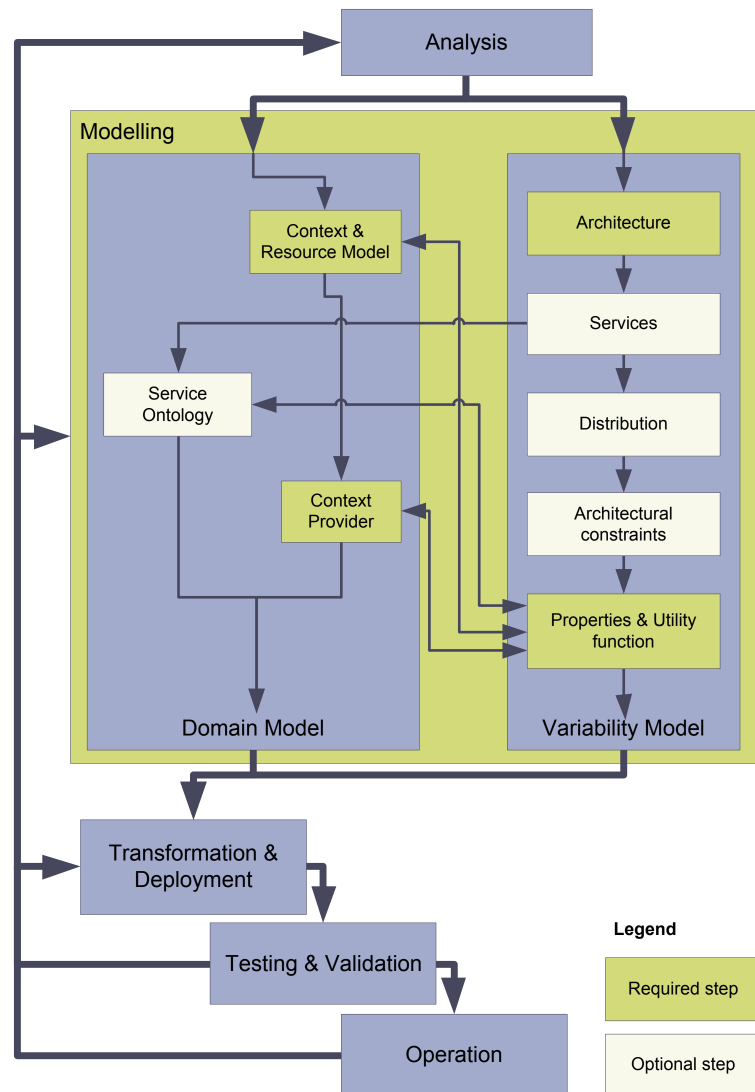


Figure 6 Overview of the MUSIC development methodology

The MUSIC methodology provides a well-documented step-by-step guideline for the development of adaptive applications. Figure 6 shows an overview. The main tasks are adaptation analysis, adaptation modelling, model transformation & deployment, and testing & validation. Each of these tasks consists of several sub-tasks, as described briefly below. A comprehensive description of the methodology is given in (Geihs et al., 2009b). More details and tutorials can be found in public project deliverables at ist-music.berlios.de.

3.2.1 Analysis

In order to be aware of possible resource and context dependencies and necessary adaptations, the application developer needs to first get a rough understanding of the execution environment. For this purpose MUSIC provides a collection of example resources and context elements that support the developer to establish an initial list of resource and context dependencies of the application. 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 the different nodes constituting the distributed execution environment and has to get an overview of potential external services available in the execution environment. On this basis, a developer performs a requirements analysis resulting in a use case model. With help of the use case model part-functionalities[4] that are influenced by changes in the execution context have to be identified.

Subtasks of the analysis are:

- build initial list of resource and context dependencies;
- identify the kind of nodes constituting the distributed execution environment;
- identify potential services available in the execution environment;
- perform use case modelling for requirements analysis.

For example, consider the InstantSocial application from the scenario. An interesting QoS characteristic of the application is the *availability,* which is a measure of the likelihood that content continues to be available. The *resource utilisation*, which is a measure of how much of the available resources are in use on the device, and the *number of instances* of the content repository component currently constituting the shared repository, is relevant context. Relevant resources for the adaptation are memory, CPU and network bandwidth.

3.2.2 Modelling

The part-functionalities identified during the analysis are the starting point for the application adaptation model which manifests the core of our adaptation approach. The application developer is expected to build the adaptation model. It captures the application variability, i.e. it specifies alternative architectural configurations for the application and how different realizations for the part-functionalities can be configured. This is called compositional adaptation. In addition, the adaptation model can express other types of adaptations such as parametric and deployment adaptation as well as adaptation by binding to an external service.

The adaptation model specifies all possible application variants and their context dependencies. Furthermore, it contains a utility function that is used by the middleware at run-time to compute the utilities of different application configurations. One of the fundamental assumptions of the MUSIC project was that ubiquitous computing environments are inherently dynamic and heterogeneous. This applies in particular to context sensors and external service offers. New sensors and service instances may be discovered while others disappear; they may be semantically equivalent but their syntactical interfaces may vary substantially; data representations may be different and conversion routines are needed in order to map values from one representation to another. In other words, middleware and applications have to cope with the dynamics and heterogeneity. Bridging the heterogeneity and reasoning about compatibilities requires a domain model that captures the relevant domain knowledge and enables

[4] With part-functionality we refer to an application function that would typically be realized by a component as part of the application's component composition.

reasoning about the relationships of domain artefacts. Therefore, the model-driven approach of MUSIC incorporates a domain ontology which serves exactly these purposes. In the current version of the MUSIC middleware prototype the ontology is used to generate on-the-fly conversion plug-ins for heterogeneous data representations of context sensor data and service parameters, possibly building a chain of successive conversions if there is no direct conversion available. More elaborate mismatch repairs, such as mapping semantically equivalent service types, are conceivable but have not been implemented so far.

Subtasks of the modelling step are:

- build a domain model that provides semantic information on the execution domain with respect to the context and resource entities as well as the service landscape; this comes into play during model transformation and application runtime;
- build an adaptation model that specifies application variants and their dependencies on the context and resources.

In summary, the main research innovations and contributions of the MUSIC modelling approach are a comprehensive application adaptation modelling methodology that enables various types of application variability and seamlessly integrates into the adaptation models information on context dependencies (as described above), objective functions (based on utility functions) and external heterogeneous service offers (that can be dynamically detected and used possibly instead of application components). The adaptation model, augmented by information from the domain model, is then processed by a model transformation, as described in the following section.

3.2.3 Model Transformation and Deployment

With the help of appropriate transformation tools, source code is generated from the adaptation model that publishes the adaptation capabilities of an application, context artefacts and data-structures for resource and context elements to the middleware. The platform-independent application variability model and the domain model serve as input to the model transformation. In general, it is not possible to completely model utility functions and property predictors using UML. Therefore, we do not intend to automatically generate the complete source code for such functions – except for property predictors for performance properties that can be covered by the property prediction framework of MUSIC and can be evaluated completely with the help of a minimal set of specifications. For the remaining property predictors and the utility function, only container classes will be generated. The application developer has to fill out the “gaps” in the generated source code fragments. The transformation tool also generates component skeletons automatically.

To enable the application to run on the MUSIC middleware the application has to be packaged in and deployed on the target device. After the necessary components and context sensors are available, the developer proceeds to create the package that will be deployed on the middleware. In summary, subtasks of transformation and deployment are:

- build model transformations that generate application code fragments;
- generate source code fragments for utility functions and property evaluators;
- manage the packaging and deployment on the target device.

3.2.4 Testing and Validating

After the application has been deployed on the target device it has to be tested if it adapts to changes in the execution context in the desired manner. For this purpose the application developer may use a context simulator that is part of the MUSIC tool set, i.e. the so-called MUSIC Studio.

This involves the following subtasks:

- testing the application’s functionality without adaptations;
- simulating context changes and testing the application’s adaptation behaviour.

The development of adaptive applications cannot be done following a strict waterfall model. Instead we recommend an iterative development approach. In particular when creating the different models, it is very likely that

there will be a need to revisit models that were built in earlier steps. Following an iterative and incremental development approach the models will be refined and extended in several iterations.

3.3 Middleware architecture

An overview of the MUSIC Middleware architecture is depicted in Figure 7 including its adaptation control loop. As will be discussed below, this loop can be seen as an instantiation of the MAPE-K model, which has already been introduced earlier in this section, for the domain of mobile and ubiquitous computing. The figure shows the main components of the middleware, represented as rounded rectangles, and their mapping to the elements of the MAPE-K model. The managed system includes the applications, as indicated by the arrows connecting the MAPE loop to the application layer, a well as the computing and communication infrastructure they are executing on and the context in which they are used, as indicated by the arrows connecting the loop to the system services layer.

A main goal of the MUSIC project was to provide a widely applicable framework for the development of adaptive systems. Therefore an important concern during the design of the middleware architecture was to allow the customisation of the framework to application or application domain specific needs and to different execution environments, both hardware and software wise. This is achieved by adopting a layered architecture in combination with plug-ins. The lower layers provide abstraction from the underlying technology. The built-in components are replaced by--or complemented with--technology specific plug-ins. The latter plug into well-defined interfaces in the middleware and can be provided as part of applications. Plug-ins are shown in the figure as square rectangles.

Figure 7: MUSIC middleware

3.3.1 Kernel

The *Kernel* derives its name from the Microkernel pattern (Buschmann et al., 1996) and implements an abstraction of the underlying component model of the software being managed, including basic operations for creating, killing, connecting and dis-connecting components. It can be seen as a substrate on which the MUSIC framework is instantiated on a device. The Kernel addresses two major concerns, namely to enable the management of applications built on different component models, and to enable the dynamic (re-)configuration of the middleware itself. The component model specific parts of the Kernel are isolated in a *Capsule*, and it may embed several capsules at a time, supporting different component models.

3.3.2 System services

The *System services* component abstracts the computing and communication infrastructure, and supports its use by both middleware and application components, as well as access to its state and its configuration capabilities. More specifically the System Services component is responsible for

- Discovering other devices and the services they provide,
- Binding to and communicating with remote components and services,
- Publishing local components and services and allowing clients on remote nodes to use them. These capabilities enable both distributed applications and the distribution of the MUSIC middleware.
- Monitoring of the state of the resources of the hosting device and its communication infrastructure
- Access to the state of and to configuring the resources of the master node and slave nodes in the adaptation domain.

The design of the system services uses the plug-in mechanism to enable easy integration with different underlying technologies, for example SOAP or RMI for remote binding, and UPnP or SLP for discovery, and to interface with native APIs to access the resources on a device, for example the CPU, the memory or the battery level. Provided with necessary plug-ins for standard protocols, the adaptation middleware may take into account both services provided by other MUSIC nodes and services provided by non-MUSIC nodes (i.e. nodes not running the MUSIC middleware) as eligible bindings for service dependencies.

Providing access both to the current state of and operations to configure devices, the system services can be seen as participating both in the monitoring and execution tasks of the MAPE-K model. In order to simplify the development and use of plug-ins, we chose to group all aspects of a resource or technology in one plug-in rather than separating the monitoring and execution aspects as in the MAPE-K model.

The *Bundle manager* is responsible for defining the set of applications to be managed by a MUSIC middleware instance, and the composition plans and component variants available for assembling them. This can be done manually by a user through the GUI component, or automatically by other applications through the Bundle manager interface. The deployment unit, called bundle, is a JAR file that may contain both code and model elements. A MUSIC bundle is a flexible deployment unit, which allows bundling and deploying individual components and composition plans as well as full applications, thus enabling downloading of meta-information (plans) separately from code. When bundles are installed or removed on a node, this will be detected by the middleware, and will trigger updating of the affected models and possibly adaptation. This provides powerful support for component reuse and evolution of deployed applications and components.

The *GUI* component implements a graphical user interface for managing the MUSIC middleware and the adaptive applications. It allows the user to install and remove bundles using the Bundle Manager, to start or stop applications using the Adaptation middleware, and to edit the user profile using the Context middleware.

3.3.3 Context Middleware

The *Context middleware* is responsible for monitoring the context and detecting and analysing context changes, as well as providing access to context information both to other components of the MUSIC Middleware and to the applications. It encapsulates the diversity of context information and maintains the *Context model,* storing and

providing uniform access both to the current state and history. Thus, with respect to the MAPE-K model, the Context middleware can be viewed as covering both the M and part of the A tasks, and maintaining part of the K.

Context is understood in a wide sense and includes both the software being managed and the hardware and network infrastructure it executes on, as well as the user and its surroundings. The state of the hardware and software infrastructure is monitored by the System Services, but made available as context information by the context middleware.

When the Context middleware detects a change that might necessitate adaptation, it triggers the adaptation middleware to figure out and implement an appropriate reaction. The Adaptation middleware notifies the Context middleware about what context information is relevant for the adaptation of the currently running application. For instance, changes to context properties mentioned in utility functions or property predictors of deployed applications cause a re-evaluation, and possibly adaptation of these applications. Thus, the context-aware behaviour of the applications is realized by the coordinated work of both the Context middleware and the Adaptation middleware (described below).

The Context middleware is designed as a pluggable architecture, where individual context plug-ins are modelled as independent components, explicitly defining their runtime dependencies on each other via context types. This architectural design is motivated by the need to cope with the openness and dynamic nature of the envisaged ubiquitous computing environment. It supports the on-the-fly integration of newly discovered sensors, following a new "sensor as a service" design principle. The activation of the plug-ins is implemented using an automated mechanism which monitors the varying context needs of the applications and starts and/or stops the plug-ins accordingly, thus achieving significant resource savings (Paspallis et al., 2008).

There are two kinds of context plug-ins, *Context sensors* and *Context reasoners*. Context sensors monitor directly observable context, for example reading hardware sensor data, while Context reasoners analyses data from several Context sensors or other Context reasoners and derive higher level context information. For example, a Context reasoner could trigger adaptation planning only when there is major differentiation in the network signal strength *trend* (as it is computed by the corresponding context plug-ins) rather than on every small change in the signal strength. This leads to both cleaner code — in terms of simpler, more expressive utility functions — as well as more efficient runtime operation — in terms of less workload in the Adaptation middleware.

3.3.3 Adaptation Middleware

The *Adaptation Middleware* is responsible for adapting the software in response to changes in the context, as detected by the context middleware, or by changes in the availability of component realisations or external services, detected by the Kernel or the System Services. It groups middleware components depending heavily on the *Adaptation model,* which is a runtime representation of the annotated architecture model provided by the application developer. The Adaptation model is represented as *types* and *plans* within the middleware. A plan has a type and represents a realization of this type, and thus a possible binding of a role or a port with a matching type. There are different kinds of plans representing different variability mechanisms. An atomic plan represents a realisation of an atomic component and describes its ports and associated property predictors, implicit dependencies on the hosting platform (e.g., platform type and version), and a reference to the code, which realizes the component. A composition plan represents an abstract composition of components. Also in this case the plan describes its ports and associated property predictors, but in addition the internal structure in terms of roles and ports and the connections between them. A variant plan represents a possible setting of the configuration parameters of a parameterised component.

The Adaptation Controller coordinates the adaptation and basically implements the A, P and E tasks of the MAPE-K model. It first calls the *Adaptation reasoner* to search the set of possible configurations of the managed system for the best fit with the current context (A) . If this is different from the current configuration it invokes the Configuration Planner is to derive a script for the reconfiguration to the preferred configuration (P). The script is handed over to the *Configuration executor*, which executes the script using operations offered by the System services and the Kernel (E)

In order to avoid that an adaptation is immediately followed by a new adaptation, the Adaptation Controller will be suspended for a short period of time after an adaptation has been performed. The length of this suspension period is configurable at runtime.

Unlike in most other systems where the reasoning method is fixed, the reasoning algorithm in the MUSIC middleware is a pluggable component, allowing the tailoring of the middleware to different needs. We have implemented several variants with different properties: i) The "brute force" variant combines the models of all running applications into one model and enumerates all possible configurations. This ensures always an optimal configuration, but scales poorly due to the combinatorial explosion of the number of configurations. ii) The "serene greedy" variant (Scholz and Mehlhase, 2010) reasons separately on each application and assumes that resources are shared between applications following the "greedy" principle. It also supresses reasoning on applications which are not directly affected by the triggering change. This algorithm performs and scales much better, but when resources are scarce, it will not always find the optimal solution. iii) The adaptation reasoning can be delegated to a more powerful remote node in order to speed up the computation and reduce the load on the master node. iv) On PCs or servers it is also possible to use a reasoner based on constraint programming, where the adaptation model is rewritten as a constraint programming problem which is solved by a general purpose constraint solver (see Section 5.2 for further details regarding performance and scalability).

It has also been considered to translate the model into an extensional representation of the mapping from context to configuration, enabling a much more efficient lookup of the optimal configuration in a given context. However, this would require the re-computation of the mapping when variants appear and disappear. As such events are assumed to be quite frequent in the kind of environments we are targeting, this idea was discarded.

Replacing or relocating components at runtime requires to manage their lifecycle with operations to stop and launch the components, to connect and disconnect them, and to set configuration parameters. Every application component must implement an interface to support this lifecycle according to the application requirements. For example, if the application needs to maintain state, the developer must implement this interface in a way that ensures the retrieval and storage of the component state. This interface and lifecycle is based on the Component configurator pattern (Schmidt et al., 2000).

In the case of a distributed adaptation domain, an adaptation may require reconfiguration both on the master node and on the slaves. In this case the Configuration Planner splits the script into batches for each node and the Adaptation Controller distributes them to the local Configuration executor on each node. This design was chosen in order to avoid excessive use of remote calls, which are known to be far more costly than local calls.

Slaves may serve several masters. To manage the sharing of the resources of the slaves a special protocol is used. The adaptation controller will query the slaves about the resources they are willing to share before invoking the adaptation reasoning, and after the completion of the reconfiguration it will inform the slaves about the estimated change in estimated resource usage. A slave will only engage in this protocol with one master at a time. In the current implementations, the slaves share their resources according to a first come first serve policy. However, as long as they adhere to the protocol, they may use any policy they want.

The *SLA manager* implements a custom, lightweight, MUSIC-specific SLA negotiation and agreement framework that enables SLA negotiation among MUSIC nodes. Together with the ability of the service publishing and discovery mechanisms to support QoS properties, this facilitates the coordinated adaptation of service based systems distributed on several adaptation domains as discussed in section 3.1. Even if it is specific to MUSIC, the negotiation framework is extensible based on the plug-in mechanism, which provides means for adding specific client-side negotiation capabilities based on alternative protocols. In particular, with the purpose of negotiating with external, non-MUSIC service providers, the MUSIC middleware can be extended with client plug-ins which comply to a specific standard, e.g. WS-Agreement (Andrieux et al., 2007).

The *Profile Manager* manages distributed adaptation domains, where a master device also controls the adaptation of a number of slave nodes. Nodes are set up manually as master or slave at start-up of the middleware or at runtime by means of the GUI (see below). Slave nodes publish a service that allows the master to discover them, access the state of their resources, and instantiate components on them. The Profile manager on the master

dynamically selects the slaves to include in the adaptation domain. The selection algorithm is implemented by a plug-in, the *Profile reasoner*, allowing application developers to provide an application domain specific algorithm. The default algorithm is to include all discovered slaves. The actual deployment of components is decided by the adaptation reasoner based on the adaptation model.

4. IMPLEMENTATION

Proof of concept prototypes were considered an important and indispensable part of the MUSIC work programme in order to show the feasibility of the conceptual design, as described in the previous section. This section discusses implementation aspects of the MUSIC framework and methodology.

4.1 Modelling language

The Unified Modelling Language (UML) was chosen as a foundation for the adaptation modelling. A newly developed MUSIC UML Profile provides a range of stereotypes and associated tagged values to specify the application architecture, its variation points, component and service properties, their context dependencies and the links between the domain model and the adaptation model. The profile contains three packages extending the UML2.1 meta-model, i.e. *Types, Realizations,* and *MiddlewareExtensions*. In the following we briefly describe these packages, and for one package, i.e. for *Realizations*, we provide a diagram including more detailed explanations. This should suffice to understand the principles of our modelling language approach. Further details can be found in the deliverables of Work Package 6 at ist-music.berlios.de.

Types: This package contains modelling concepts for component type, property type, application type and service type. They are modelled as UML classes. For a property type it is important to know whether it refers to a resource or a context entity. A specific *isResourcePropertyType* tag is used as an indication. Finally, an application type is a specialization of a component type.

Realizations: An excerpt of the UML profile for modelling realizations is shown in Figure 8. The stereotype *«mRealization»* is used in the profile as super-class of the realization stereotypes, but it will never be used explicitly in an adaptation model. The realization details of a component type or an application type are contained in the stereotypes *«mAtomicRealization»* or *«mCompositeRealization»*. A composite realization represents a variation point for the adaptation reasoning, as explained above in Section 3.

In case of a type realized by a discovered service, a different set of information is needed and declared in the stereotype *«mServiceInfo»*. Moreover, in case of parametric adaptation the realization may vary depending on the value of some application parameter. This is modelled in stereotype *«mRealizationVariant»* specifying the range of parameter values. Deployment of components to nodes can also be specified using a stereotype *«mRoleDeployment»*, which is not shown in the diagram for the sake of clarity.

A property predictor is modelled as a UML class, where references to the related property type, involved context entities, role, etc. are included as tagged values. A utility function is modelled as a specialization of property predictor, whereby the value of the *propertyType* tag is ignored. Note that the model transformation will generate only source code fragments for property predictors and utility functions. These fragments need to be completed manually. To facilitate this step, pseudo-code and implementation guidelines can be inserted as a comment into the model.

A context query is associated with a context-related property type. Such queries can be modelled both as UML classes and as UML comments. A resource query is associated with a resource-related property type. Tagged values are used to specify the context/resource entity, scope of the context information, its representation etc.

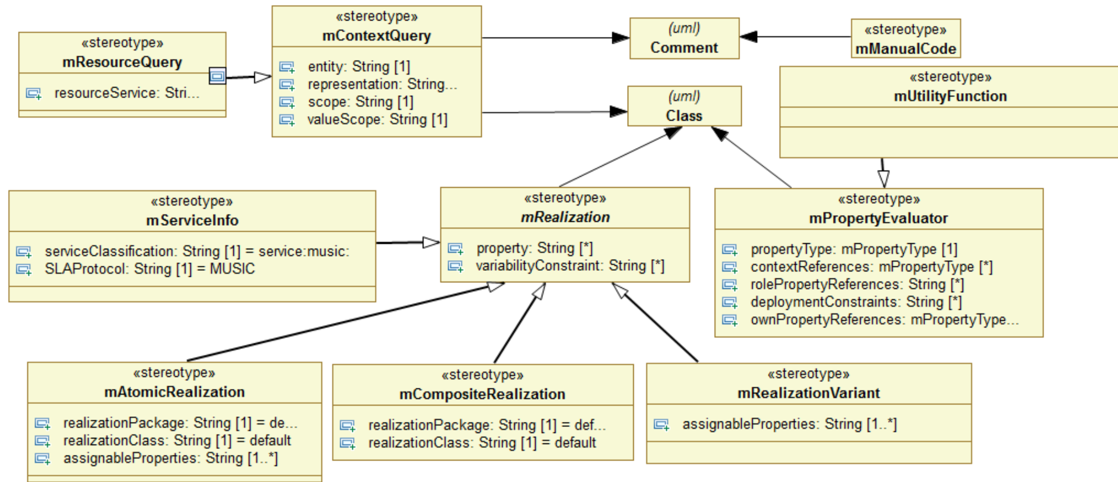


Figure 8: Excerpt of the UML profile for modelling realizations

Middleware extensions: In the current version of the modelling language this package mainly supports the modelling of context plug-ins that encapsulate context sensors and reasoners. If more self-adaptation features of the middleware itself were to be added to the MUSIC framework, this package would be extended by the corresponding modelling concepts.

4.2 Tools

In order to take full advantage of model-driven development, appropriate tool support is a must. Tools are required to create the application adaptation model, generate source code, test and validate etc. The MUSIC Studio is a suite of tools integrated together to help the application developer in creating adaptive applications based on the MUSIC middleware. This integrated tool environment is another innovative contribution of the MUSIC project. The suite contains a mix of selected pre-existing (preferably) open source tools as well as custom developed tools for the MUSIC project. The MUSIC Studio is embedded into the Eclipse development environment and exploits EMF and several related tools. Figure 9 shows the constituents of the MUSIC Studio.

The MUSIC Studio project environment assists the developer in setting up a project for developing MUSIC applications and components, and includes templates and wizards which set up all required files for a project involving the full tool chain of the MUSIC Studio.

The *Modelling* tool is needed for creating and editing UML models of MUSIC applications, based on the newly developed MUSIC UML profile. Many UML tools could be used here. However, in accordance with the general open source policy of MUSIC, the open source modelling tool Papyrus is preferred.

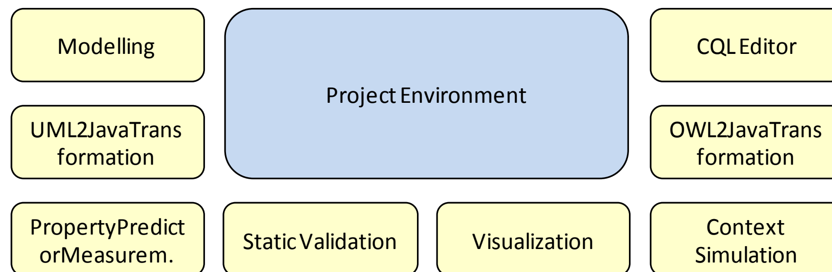


Figure 9: MUSIC Studio Overview

The *CQL Editor* provides editing support for the Context Query Language (CQL) developed as part of the context management infrastructure. CQL is an XML-based language whose syntax is described by an XML Schema Definition (XSD). This means that most existing XML editors that understand XSDs, can provide basic CQL support.

The *UML2JavaTransformation* tool transforms variability models created with the Modelling tool to a representation useable by the MUSIC middleware. We employ the widely used MOFScript transformation builder to develop transformation scripts for the transformation of the application variability model to Java source code.

The *OWL2JavaTransformation* tool transforms the MUSIC Service Ontology, expressed in OWL, to Java code (see figure 9 above). In general, there is a choice of suitable third party (open source) tools available for this transformation. In any case, they need to be customized for the MUSIC modelling and transformation requirements.

The MUSIC *Static Validation* tool checks MUSIC application models in order to detect errors and omissions. The main goal is to ensure that developers have filled in what is needed for a working adaptation model. It helps catching design errors that are introduced in the code and would otherwise be manifested during 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, and whether required files are correctly defined. Moreover, it may be needed to enable early testing of the values of property predictor functions. There is a special tool for this, called the *PropertyPredictorMeasurement* tool.

The *Context Simulation* tool is part of a prototype test and simulation environment that enables developers to observe and analyze the effects of context changes and adaptations, and so carry out adaptation tuning. A *Visualization* tool provides easy to grasp visual information on the state of the middleware and applications and their operations.

4.3 Middleware implementation

The reference implementation of the MUSIC middleware was built upon readily available technologies, thus letting the consortium focus on implementing the distinctive features of the project. In particular, *Java* and the *OSGi* component framework were selected as the foundation of the prototype implementation in order to meet the most relevant requirements for the MUSIC implementation:

- *Open source implementation* aligned with a general open source strategy for the MUSIC project.
- *Suitability for resource-constrained devices,* i.e. embedded and mobile devices.
- *Multi-platform support for a* range of devices (smartphones, PDAs, laptops, PCs) and operating systems.
- *SOA support* to dynamically discover and integrate services.

The Java version J2SE 1.4 was selected as a base but it is constrained to the common subset with Java ME CDC in order to assure compatibility of the implementation with both Java frameworks. MUSIC is also available on Android, supporting the Dalvik VM.

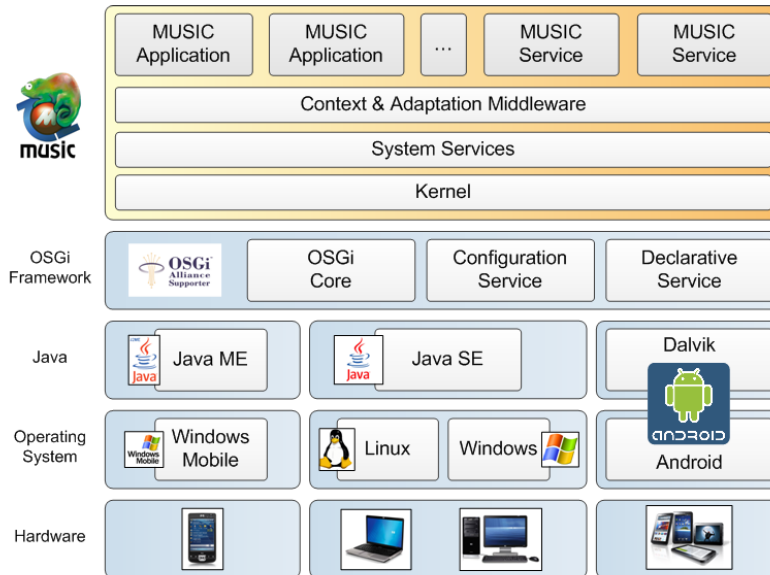


Figure 10 The reference implementation of the MUSIC middleware

OSGi has established itself as the dynamic module system for Java. It is a service-oriented component framework. The success of OSGi may be attributed to its relative simplicity, efficiency, openness, and portability. In particular, OSGi

- is a standard, consolidated (R1 in 2000) and in constant evolution (latest revision is R4.3 from September 2011),
- was designed for resource-constrained devices,
- offers a service-oriented, component-based middleware,
- supports deployment on-the-fly,
- provides remote management,
- includes a useful set of standard services,
- is available in several open source implementations.

OSGi offers a class-loading mechanism to dynamically load/unload modules (bundles in the OSGi terminology). This feature is particularly interesting to support the plug-ability of the MUSIC architecture. Plug-ability is required to tackle the heterogeneity in communication and service discovery technologies. It also allows the integration of an extensible set of customized context sensors and adaptation algorithms.

The MUSIC middleware is structured in different components (bundles) collaborating together by importing and exporting Java packages, and by providing and consuming services. The middleware can be deployed in a heterogeneous execution environment on top of OSGi as depicted in

Figure 10. The portability of the reference implementation is based on the Java slogan "*Write once, run anywhere*". MUSIC also tries to minimize the mobile fragmentation that complicates the development of applications in mobile devices. In fact, MUSIC offers a common framework for mobile devices and non-mobile devices.

The reference implementation can be hosted by any device supporting Java 1.4, Java ME CDC (e.g. phoneME Advanced) or Android. Especially the advent of Android OS turned out to be a great opportunity to extend the installation base for MUSIC. The middleware and two applications are now available for free in the Android market.

In order to be widely acceptable and stimulate the largest possible uptake, all MUSIC reference implementations are based on open standards, implemented in Java and released under an open source license not

preventing commercial reuse, while requiring new extensions to the MUSIC framework itself to be released as open source. In this way, the development of the framework will continue, driven by its users, long after the end of the MUSIC project.

5 EVALUATION

The implementation of the MUSIC development framework described in the previous section has been used for an evaluation of our approach based on trial development and testing of the developed adaptive software. In this section we describe the trials and discuss the findings in relation to the challenges stated in Chapter 2.

5.1 Experimentation

In an effort to confirm the values of the MUSIC development methodology and framework, we used both an internal and an external experimental evaluation approach. In addition the framework has been used in several follow-on projects by different MUSIC partners.

The internal experiment included the development and testing of a collection of trial applications by developers from the MUSIC partner organizations. The collection of trial application includes the TravelAssistant and the InstantSocial applications used as examples in this paper. In addition there is the PRM, the Conferencing, the FirstAid and the Satcom applications. PRM is an assistant for people with reduced mobility requiring assistance by metro staff to get on and off the train. It allows the staff to track the location of the passenger and to exchange messages with them. It demonstrates the context distribution capabilities of the framework, letting the passengers share their location with metro staff. Conferencing and FirstAid is a pair of applications aiming to assist when passengers are injured or suffer indisposition during travel and where medical first aid is required. It assists both with first aid instruction and with establishing contact with the metro staff and with locating and establishing contact with people with relevant medical competence in the vicinity. Satcom is an application assisting satellite antenna installers, with aligning the antennas, dispatching jobs to installers, and navigation to customer sites.

The experience of the developers regarding the use of the modelling language and tools was collected using questionnaires. Nine developers with experience ranging from 4 to 13 years participated. The functioning and performance of the middleware were tested in the lab running test scenarios realised through simulated context changes. During the tests the correctness of the adaptation behaviour was checked and data for various performance parameters were collected and analysed. At the end of the project the trial applications were demonstrated successfully to a group of journalists in and around the Paris Metro during regular day-time operating hours. The journalists participated in the demo by using the trial application on devices handed out to them.

In addition to the trial applications, a few artificial applications were created to test certain aspects of performance and scalability. Both in the lab tests and the demonstration we used the "serene greedy" variant of the planning algorithm.

Further details regarding these experiments and the demo can be found in public project deliverables at ist-music.berlios.de.

One external experiment was done at the University of Cyprus. It included undergraduate students who used MUSIC as part of a ubiquitous computing course. Students were first given an introduction to the OSGi framework, development of context aware applications using “brute force” programming, and — finally — use of (a subset of) MUSIC, all in approximately 10 weekly sessions of 90 minutes each.

After covering basic OSGi concepts, the students were instructed to develop context-aware applications using so-called “brute force” programming, i.e. developing using just the core language features (provided by Java itself, as well as the OSGi framework). Then, in the last group of sessions, the students were tutored the basic functionality of MUSIC with emphasis on building and utilizing context plug-in bundles. At that point, the students were asked to re-implement their context-aware applications. This second step included the option of reusing context plug-ins available in the MUSIC repository or provided by other students, highlighting the middleware’s potential for code

reuse. Last, their applications were finalized (i.e. by adding the UI and business logic), and were bound to the middleware for asynchronous notification of context changes.

The evaluation was completed in the form of questionnaires, asking the developers and the students who got hands-on-experience with MUSIC to report their experience. The format of the questionnaires, as well as the cumulative answers received, are available in appendix B of (Paspallis, 2009).

At the University of Kassel the framework has been adopted and used in a follow-up research project by application developers who were not involved in the MUSIC project. The objective was to develop a prototype of a new adaptive mobile social networking application for Android phones on top of the MUSIC middleware. Five groups of undergraduate and graduate students, advised by experienced MUSIC developers, worked on different variants of the application during a semester project. Together they built the adaptation model. At the end of the semester questionnaires and interviews were used to find out about the experiences with the development framework and methodology.

In a project at SINTEF the MUSIC technology is being used as a central component in the development of an experimental development platform for internet-of-things applications, with a special focus on simplifying the deployment and evolution of such applications. The idea is to exploit the self-adaptation and extensibility capabilities of MUSIC to build self-configuring and extensible software which adapts semi-automatically as things, services and applications enter and leave the network. Application areas considered include ambient assisted living and sub-sea sensor networks using acoustic communication.

5.2 Findings

This section reports our findings, as they were identified from the different experiments conducted as part of the MUSIC project. Since the different experiments provided mostly consistent results, the overall findings are reported here - with discussions of significant deviations, where appropriate.

5.2.1 Learning and using the methodology, modelling notation and tools

Throughout its development, MUSIC was steered by the guidelines summarized at the end of section 2. Most notably, it aimed at making it easier for developers to produce and maintain large and complex context-aware, self-adaptive applications. The findings from the experiments indicate that:

- The framework provides very comprehensive and effective development support while also removing a substantial amount of work from the shoulders of the developers.
- The tight alignment of the methodology with the middleware works very well.
- The flexible plugin-architecture of the context and the adaptation middleware shows clear advantages when individual applications require their own specific solutions (e.g. specialized context sensors and reasoners, communication protocols, or adaptation reasoners).
- The integration into Eclipse has proved successful and effective.
- The MUSIC demonstrators are helpful in getting acquainted with the framework and help serve as blueprints for new applications.

For every research endeavour, discovering the drawbacks of a certain methodology, model or tool is equally important as finding its advantages. As such, the experimentation has helped us identify the following problems and opportunities for improvement:

- The MUSIC framework is quite extensive and thus has a *steep learning curve*. For instance, newcomers had to spend a lot of time familiarizing themselves with many necessary concepts before they were able to develop their own “hello world” application. Providing tutorials helped to mitigate this problem, but the latter are limited to explaining the methodology, whereas limited information is provided on the handling of the framework. A more elaborate user manual would help overcome this drawback.

- Many developers have questioned the effort spent on adaptation modelling because most applications have only a few variants that might as well be handled without a sophisticated adaptation modelling environment. Ideally, a solution like MUSIC should scale its complexity alongside the complexity of the underlying problem (e.g. the developed application). The complexity of modelling is partly due to limitations in the UML profiling capability, and the problem could be mitigated by providing a DSL (domain specific language) more like the abstract notation used in the examples in this paper, and a special purpose modelling tool.
- As an open-source project, MUSIC focused its effort on building upon other public domain solutions. However, that turned out to cause problems in some cases because these components were not mature enough. An example is the Papyrus UML tool which caused some frustrations among the developers and led to modelling mistakes and incorrect UML models, and consequently to incorrect model transformations. A more mature UML tool would have provided a better experience and a lower entrance burden, or even better, a DSL with its own tool as mentioned above.
- Using the framework assumes good know-how of the underlying OSGi component framework in order to fully exploit the MUSIC potential. As OSGi is becoming more popular in some domains, this should become a minor problem. Nevertheless the additional OSGi layer adds to the complexity of the middleware (for instance debugging OSGi-based software can often be challenging).

Besides the above common findings, a more specialized result refers to the experiment at the University of Cyprus. That experiment was limited to the context middleware part of MUSIC. The feedback from the students was favourable for the presented development methodology and underlying framework (compared to the so-called “brute force” approach). In particular, 11 of 12 students stated that they would like to use the MUSIC framework — as opposed to a brute-force approach — had they been assigned the development of additional context-aware applications. Similar answers were received also by the internal developers who completed the questionnaires. It could be argued that the main finding from this exercise is that it has been proved that building context-aware applications adds significant complexity (as expected) and also that providing a methodology and an appropriate middleware can help improve the capacity and ability of the developers.

In general, it turned out that the learning process for the MUSIC framework initially requires a rather large effort and is hindered by several more or less severe obstacles, mostly due to issues related to the handling of tools and middleware. However, after having climbed the initial hurdle the development methodology and middleware infrastructure as such were found to be very appropriate and effective for the development and operation of complex context-aware, self-adaptive applications.

5.2.2 Correctness and performance at runtime

Generally, the testing of the trial applications confirmed that the envisaged adaptation behaviour was achieved, although in most cases some tuning of the context reasoners, property predictors, and utility function during testing was necessary. In the Kassel experiment, which took place after the MUSIC project had finished, it was observed that the quality of the middleware prototype is very good; no severe bugs were detected by the students.

Some results from the measurement of the processing overhead for the adaptation planning and reconfiguration are shown in Table 1. The measured application is TravelAssistant, and the planning algorithm is the "serene greedy" variant. First we measured TravelAssistant running alone and then with InstantSocial running in the background. A sequence of events roughly corresponding to what happens in the scenario was simulated. The tests were performed on a Google Android G1 smartphone with MS Windows PCs acting as slave and service provider nodes, all connected in a WLAN. All measurements are given in milliseconds. Note that the measurements are done by means of logging code, which according to our experience degrades the performance by 20-50%. Thus the overhead experienced by a user will be 20-50% less than what is measured.

During adaptation planning, the application still runs. However it competes with the middleware for the CPU, so its responsiveness may suffer. As can be seen from the measurements this typically lasts for a few seconds each time an adaptation planning is triggered. When adaptation planning has decided on an adaptation, the application is paused while reconfiguration takes place. This also typically lasts a few seconds.

Table 1 Example performance measurements

			TravelAssistant alone			Travel Assistant with InstantSocial in the background		
	Event	#vari-ants	Planning (ms)	Reconfig (ms)	Total (ms)	Planning (ms)	Reconfig. (ms)	Total (ms)
1	Launching Travel Assistant	8	3864,0	2815,8	6679,8	2887,6	2467,6	5355,2
2	Context change that causes a reconfiguration	8	3098,0	504,4	3602,4	3047,2	581,0	3628,2
3	Context change that doesn't cause a reconfiguration	8	3006,6	39,4	3046,0	3505,0	102,8	3607,8
4	Discovery of services (service binding)	32	2933,8	325,2	3259,0	3484,4	407,6	3892,0
5	Slave node also discovered (along with the services)	48	5385,0	301,6	5686,6	5365,0	379,0	5744,0
6	Disappearance of services (service unbinding)	8	4390,2	89,4	4479,6	4018,0	156,6	4174,6
7	Distributed application (instantiating component in slave node	48	6417,0	2996,8	9413,8	6701,8	2919,2	9621,0
8	Disconnecting from slave (de-instantiating component in slave node)	48	6517,2	1638,0	8155,2	6700,2	2009,0	8709,2

There is not much difference in the performance for one or two applications. This is because the context changes that affect the Travel Assistant are different from those affecting Instant Social, and heuristics applied by the planning algorithm avoids adapting applications not affected directly unless there is a shortage of resources. Similar tests with the First Aid and Conferencing applications on Windows Mobile phones of the same class gave similar results.

In the demo we also used Google NexusOne phones (1 Ghz processor, 512 MB RAM), which had then just arrived on the market. While rehearsing for the demo we did similar measurements on this phone during a trip on the Paris metro with several (up to 5) phones running both TravelAssistant and InstantSocial, and devices joining and leaving. Even if this test involved more nodes and active use of InstantSocial, the results showed that planning and reconfiguration was about halved compared to the experiment with the G1. During the course of the demo walkthrough we had 36 adaptations during nearly 2 hours of runtime. Average adaptation time was 3375 ms for adaptations with a reconfiguration and 1084 ms without. The maximum was about 9 seconds whereas the minimum was only 241ms. It is worth noting that in this rather realistic test, the fraction of the total runtime where the middleware adds a significant load on the device is small, around 1%.

The observed memory footprint of the middleware is approximately 17 MB, and a typical (in our tests) application on top of MUSIC adds approximately 12 MB. The observed start-up time for the middleware was up to approximately 45 seconds, depending on device. For example the longest time was observed on the Android G1, while it was much shorter on a Nexus One. Although this can be conceived as somewhat slow, we consider this is a lesser problem since the middleware is supposed to run continuously and started only when the device is switched on. Platforms, such as Android, can be set up to automatically start the middleware when the device is booted.

From our experiments we conclude that for today's mobile platforms the middleware is a bit heavy-weight in terms of memory demand and start-up time, and also processing overhead during application execution. However, in practice, during the demo, the overhead of the middleware did not affect the responsiveness of the application in a disturbing way.

5.2.3 Scalability

Our adaptation planning approach basically works by evaluating all possible configurations and selecting the one with the highest utility. Due to the combinatorial explosion, the computational complexity of this algorithm is exponential in the number of variation points. Thus, this approach has an inherent scalability problem. However, we observed during the development of the trial applications that each application typically did not have more variants than could be handled with acceptable performance. Performance problems occurred with the "brute force" variant of the algorithm when running several applications concurrently. We also noted that the need to coordinate the adaptation of different applications only stemmed from competition for shared resources, and that different applications tended to be affected by different context. The "serene greedy" variant of the algorithm exploits this to improve the scalability as explained in section 3.3.2. In the best case, when there is no shortage of resources and the applications are affected by disjunctive contexts, the planning time of the "serene greedy" algorithm is unaffected by the number of running applications, as it will only replan the directly affected application. In the worst case, the "serene greedy" algorithm will replan all the running applications and the planning time will be proportional to the sum of the number of variants of the running applications, rather than the product, which is the case for the "brute force" algorithm.

A comparison of the performance of the two variants of the planning algorithm is shown in Table 2. The measurements were done with several instances of a manipulated version of TravelAssistant where additional versions of the components were added artificially increasing the number of variants to 2048. The instances were launched one by one and we measured the planning time for the initial configuration of the application. This corresponds to the best case behaviour of the "serene greedy" algorithm. The test was run on a laptop, and that is the reason that the observed planning times are significantly shorter than those observed on the smartphones, although the test applications have more variants.

This shows that the scalability of the "brute force" algorithm is insufficient in multi-application scenarios, but that the "serene greedy" algorithm solves this problem.

.Another opportunity for improvement is early filtering (Brataas et al., 2007). In the trials we have seen that the utility functions often contain code to exclude certain combinations in certain contexts based on the properties of individual components or services. This may be exploited to exclude invalid variants in a more efficient way and thus reducing considerably the number of variants that has to be evaluated during adaptation planning.

We also did some experiments with the adaptation planner based on constraint programming. The solver can apply different heuristics to reduce the solving time, and for the most efficient ones the solving time is linear with the number of variation points. Admittedly, this comes at the expense of no guarantee for an optimal choice. However, in our tests the utility of the chosen variant were very close to the utility of the optimal variant (between 95 and 100 %). This test was done with artificial adaptation models, not with the trial applications. Still the results indicate that it is possible to apply search heuristics that significantly improve scalability.

Table 2 Execution times of the "brute force" and the "serene greedy" algorithms

	1st launch	2nd launch	3rd launch	4th launch
"Brute force" algorithm	78 ms	40765 ms	>20 min.	not tested
"Greedy" algorithm	156 ms	141 ms	125 ms	141ms

5.2.4 Reusability and evolvability

A prominent feature of the MUSIC approach is the modularity of the adaptation model. Components and compositions with their adaptation models can be developed separately and bundled and reused in different systems. Furthermore, already deployed and running applications can be extended or modified by deploying new bundles representing new alternative implementations at all levels of the design, and by retiring already deployed bundles. The trial applications have demonstrated that this works as intended, but since the trials only covered the initial development phase we could not really assess the value of this feature.

5.3 Threats to validity

Our evaluation of the appropriateness and learning effort of the development framework is based on developers that were participating in the project and students, and in all cases they had support from experts taking part in the development of the framework. Some of the trial application developers also participated actively in the development of the framework and thus were experts themselves. This may have biased their assessment in a positive direction. On the other hand, the assessment was based on experience with a technology under development, with immature and lacking documentation, which may have caused a bias in the other direction.

Our experiments covered only the initial development phase of the trial applications. It is widely accepted that model based development approaches and separation of concerns pays off most in later phases of the software lifecycle. The same holds for the support for component reuse and for dynamically extending and modifying both the functionality and the adaptation behaviour of the applications.

Finally there is the issue of the representativeness of the trial applications and the scenarios used for the test. There is of course the possibility that both the applications and the scenarios are too simple to really assess the applicability of the MUSIC framework. Regarding the applications, although they are fairly simple, their adaptation behaviour is not trivial and requires the consideration of several context and resource properties and the balancing of conflicting concerns. Regarding the test scenarios, we believe they are staged in a typical and challenging environment for mobile computing, and that they represent typical usage examples for mobile devices already emerging as common practice.

In conclusion we believe that our evaluation has demonstrated the feasibility of the MUSIC framework, and that it to a large extent meets important challenges related to the development of adaptive software. However, it has also identified some problem spots where the framework needs further refinement.

6 RELATED WORK

Context-aware and self-adaptive applications that are able to adapt to context changes have been the subject of many projects over the years. These projects differ in their technical approaches and scope, as well as in their application domains. In this section we compare MUSIC to a few typical representatives of the many projects that have tackled context-aware adaptation from different angles.

6.1 Context-awareness

Following Mark Weiser's introduction of his vision of Ubiquitous Computing (Weiser, 1991) context-aware computing has been actively studied since at least the mid-nineties. Initially, the research was concerned primarily with stand-alone location-aware applications (Long et al., 1996; Want et al., 1992). However, as the interest for context-awareness was extended to cover a much wider domain of context types and uses, the focus naturally shifted towards building frameworks providing support for general context-awareness.

Perhaps the most popular of those is Dey's Context Toolkit (Dey, 2001). When it was first introduced the *Context Toolkit* was a pioneering approach. The basic idea was to simplify the development of context-aware applications by enabling the reuse of specialized components, in a similar way to how widgets facilitate the development of advanced graphical user interfaces. While the Context Toolkit was based on reusable code, it was

limited in a number of respects: First, applications were assumed to integrate their own context management code, whereas in MUSIC this code is external (embedded in the middleware). Notably, this implies that the Context Toolkit could not facilitate context reuse by concurrent applications running on the same device. Second, the Context Toolkit management code was rather rigid itself, whereas in MUSIC individual context plug-ins enabled automatically starting or stopping context acquisition and processing on demand, thus optimizing resource utilization.

The *Context Information Service* (CIS) (Judd and Steenkiste, 2003), developed by and used in Aura (Sousa and Garlan, 2002), provided a database approach for context-aware applications where queries are encoded in a SQL-like language. Arguably, one of the advantages of this approach was that it delegated some of the power of traditional Database Management Systems (DBMS) to context clients, via a SQL-like query language. Nevertheless, and unlike MUSIC, CIS was limited in terms of controlling the activation and deactivation of various — distributed — context providers, making it difficult to optimize resource utilization.

The *Context Modelling Language* (CML), proposed by Henricksen and Indulska is a graphical modelling approach which was used as a tool to assist designers with the task of exploring and specifying the context requirements of a context-aware application (Henricksen and Indulska, 2006). This model defines constructs for types of context (in terms of fact types), their classification (i.e., whether it concerns sensed, static, profiled or derived context), associated quality metadata and dependencies amongst different types of context. However, while the CML-based context model is quite powerful, it is also significantly complex, making its adoption particularly difficult. Furthermore, the authors do not consider distribution of context in their requirements, and they do not discuss how it could be enabled using CML.

Finally, the authors of (da Costa et al., 2008) propose a *General Software Architecture for Ubiquitous Computing*, which defines many requirements commonly found in the literature, such as: heterogeneity, scalability, dependability, spontaneous interaction, etc. The MUSIC middleware has noted these requirements and satisfied most of them. For instance, heterogeneity is met at multiple levels: hardware heterogeneity is implicitly supported by the underlying Java/OSGi middleware (and has been practically proven by deploying applications on both desktop, windows mobile and Android devices), while context model heterogeneity is supported via an elaborate ontology-based model (Wagner et al., 2008). An elaborate discussion of these requirements in the context of MUSIC is available in (Paspallis, 2009).

6.2 Self-adaptation

McKinley et al. (McKinley et al., 2004) argued that foundations from autonomic computing along with advances in software engineering form the basis for most of the existing adaptation solutions. Furthermore, Satyanarayanan (2001) argued on how pervasive computing builds on foundations of distributed computing, shares the same requirements with mobile computing and augments both by introducing new capabilities and requirements. In this section, we present a few representative works which are related to our approach. Far more extensive surveys of self-adaptation software exist in the literature, such as (Salehie and Tahvildari, 2009) and (Kakousis et al., 2010).

As mentioned in the Introduction of this paper, the MADAM project was a direct precursor of MUSIC. MADAM addressed adaptation in mobile computing scenarios from both the theoretical and the practical perspective and provided several innovative results acting as a foundation and starting point for MUSIC (Geihs et al., 2009a). MUSIC goes far beyond MADAM in tackling ubiquitous computing environments that are characterized by inherent openness, heterogeneity, and dynamic service discovery and binding.

CARISMA (Capra et al., 2003) is a mobile computing peer-to-peer middleware that exploits the principle of reflection to support the construction of context-aware, adaptive applications. CARISMA uses a microeconomic-inspired mechanism for conflict resolution in mobile settings, which relies on a particular type of sealed bid auctions for conflict resolution while utility functions are used for incorporating user preferences in the resolution mechanism. In contrast to MUSIC, CARISMA focuses on adaptation of middleware level services. Planning in CARISMA consists of choosing among a handful of predefined rule based adaptation policies (referred to as

application profiles) using utility functions and resolving conflicts between policies using an auction-like procedure. While the adopted rule based policies are natural and simple to use, there are some drawbacks compared to MUSIC's use of utility functions. Firstly, rules provide less transparency to the developer by requiring reasoning in terms of lower-level reconfiguration actions rather than architectural design with variation points, In MUSIC, the lower level reconfiguration actions are automatically determined by the middleware. Also, adaptation in MUSIC is QoS (or property) driven while the CARISMA rule based policies do not consider prediction of non-functional properties. Furthermore, CARISMA handles only a fixed number of adaptation policies whereas the utility function approach of MUSIC allows choosing the best adaptation among an open-ended range of alternatives, both fine grained and course grained, that are automatically derived by the adaptation middleware itself. Finally, MUSIC's utility function approach allows for dynamically adding or removing components and plans without requiring a complete restructuring of the deployed applications.

Gomaa and Hussein have studied the dynamic reconfiguration problem (Gomaaa and Hussein, 2004)

The self-adaptation techniques proposed by the RAINBOW approach (Garlan et al., 2004) are similar to MUSIC in that they attempt to separate adaptation from application logic concerns. RAINBOW extends architecture-based adaptation by adopting architectural styles. Such styles are used to tailor the adaptation infrastructure by encoding the developer's system-specific knowledge, including adaptation strategies and system concerns. The resulting situation-action rules are scored using utility preferences specified for the quality dimensions, where the adaptation manager selects the highest scoring strategy. However, RAINBOW appears to have based its adaptation strategies on situation-action rules, which specify exactly what to do in certain situations. MUSIC also uses utility functions, but to score application variants. When a new variant is selected, the adaptation steps necessary to get from the current variant to the selected one are derived by the middleware.

The Aura project (Sousa and Garlan, 2002), which was built on the legacy of Odyssey and Coda, targeted primarily pervasive applications. For this reason it introduced auras (which correspond to user tasks) as first class entities. To this direction, it categorizes the techniques which support user mobility into: use of mobile devices, remote access, standard applications (ported and installed at multiple locations) and finally use of standard virtual platforms to enable mobile code to follow the user as needed. Unlike Aura, MUSIC focuses on specifying the internal architecture of mobile applications, aiming at enabling context-aware adaptive behavior via middleware support. Similar to MUSIC, Aura integrates so-called staging servers to off-load resource intensive computations from mobile devices and to improve performance when accessing remote data sources. Similarities also exist in the overall architecture where Aura defines a Context Observer, for observing and reporting context, comparable to MUSIC's Context Manager, as well as a Task Manager (PRISM) and Environment Manager, for reasoning and planning the adaptation based on the available services, comparable to MUSIC's Adaptation Manager).

Also related to Aura is research on an extensive resource property prediction model, called anticipatory configuration (Poladian et al., 2007), that adheres to the specific application, task and services model of Aura. In MUSIC we have tailored the property prediction model specifically to the compositional adaptation of context-aware component-based applications by facilitating the definition of predictor functions over the properties of the context and the involved application components, whereby we have put more emphasis on execution performance than on precision of the predictor functions.

MOCAS (*Model Of Components for Adaptive Systems*) proposed a generic state-based component model which enables the self-adaptation of software components along with their coordination (Ballagny et al., 2009). Unlike MUSIC, MOCAS focuses on the behavioural adaptation of the components themselves. Each component embeds a UML state machine to realize its behaviour at runtime. It is installed in a container managing the adaptation process and ensuring its consistency. Adaptation is triggered when invariants related to the component's business properties are violated. The component supports updates of its specification at runtime. In MUSIC such behavioural adaptations can be achieved by architectural adaptation, i.e. by binding to an appropriate component or service that exhibits the desired modified behaviour. In addition MUSIC supports other adaptation mechanisms, such as deployment adaptation and device adaptation.

Menasce and Dubey (Menasce and Dubey, 2007) propose an approach to QoS brokering in SOA. Consumers request services from a QoS broker, which selects a service provider that maximizes the consumer's utility function subject 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. This approach is of interest both from the viewpoint of the consumer and the 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 is not always possible in ubiquitous environments. Furthermore, it assumes that the consumer is able to determine the expected service properties. MUSIC, on the other hand, considers the offered properties as alternatives in order to determine the best application configuration, and in this way it enables the client to adapt to the service landscape.

Adaptive Service Grids (ASG) is a more recent open initiative that enables dynamic composition and binding of services, and which is used for provisioning adaptive services (Kuropka and Weske, 2008). Compared to the MUSIC planning-based middleware, ASG focuses only on the planning per request of service workflows with regards to the properties defined in the semantic service request. Thus, ASG does not support a uniform planning of both components and services as MUSIC does.

Finally, the MOdel-based SElf-adaptation of SOA systems (MOSES) (Cardellini et al., 2011) approach aims at the intersection of SOA systems and self-adaptation. At its core, and similar to many other approaches (including MUSIC), MOSES follows the MAPE-K reference model, using utility functions for adaptation decisions. Notably, and also similar to MUSIC, MOSES exploits the flexibility of its utility-based approach to allow for adapting based on the common benefit of all the users. However, and unlike MUSIC, as MOSES is a SOA-centric approach it is limited to service-level only adaptation and, also, its adaptation logic is limited to a predefined set of attributes only (i.e. response time, expected execution cost, and expected reliability).

6.3 Architectures for self-adaptation

As explained above, the architecture of MUSIC includes an adaptation control loop to perform self-adaptation tasks. Two, by now widely accepted architectural models for self-adaptive software systems, are the MAPE-K model (Kephart and Chess, 2003) and the three layer model of Kramer and Magee (Kramer and Magee, 2007). As evidenced in Section 3, MUSIC can be seen as an instantiation of the MAPE-K model for mobile and ubiquitous environments.

Kramer and Magee suggests a layered reference model for self-adaptive software systems with the three layers Component Management, Change Management and Goal Management respectively. The Component Management layer implement operations to monitor and to manipulate the managed system, the Change Management layer execute predefined reconfiguration plans for foreseen situations, while the Goal management layer manages goals and derives new plans for situations where no relevant plans are available.

The above three layered model was initially not considered for MUSIC as this model was published after the basic architectural principles of MUSIC was fixed and basically adopted from the MADAM project (Floch et al, 2006). Nevertheless, it can be observed that the activities of the MUSIC adaptation middleware correspond to activities in the change management layer of the Kramer & Magee model, while the rest of the MUSIC middleware, and the application component roughly correspond to activities found in the component management layer.

Considering change management, MUSIC adaptation models seem to roughly correspond to application plans in the Kramer and Magee model in the sense that alternative application configurations (application variants) can be derived from a single application model. A difference in the layering is that the MUSIC adaptation middleware does not assume there can be local control loops within application components. Instead all adaptation decisions are made by the adaptation middleware, including tuning parameterized behaviour. We recognize, however that local adaptations may help boost the performance of adaptive software, and the support for this by MUSIC, e,g, in the form of hierarchical adaptation loops, would be useful.

MUSIC does not include anything corresponding to the Goal Management Layer in the sense that MUSIC is not able to autonomously generate new or updated plans (in the form of application models) in response to planning failure. However, adaptation models can be updated manually in an incremental manner by deploying or removing plans and component realisations, thereby modifying or extending the models of running application. New variant models will be considered in future adaptation reasoning. Also, the starting and stopping of applications on the mobile device will result in the addition and removal of plans in the adaptation model and changing the overall goal by the addition and removal of utility functions. These tasks could be considered as logically belonging to goal management.

Considering the MUSIC context middleware, its function and internal structure belong to the Component Management layer of the Kramer and Magee model since apparently storing and providing access to the state of hardware and software components as well as the monitoring of the state of the surroundings of the user, such as ambient light and noise, and user preferences, belong to this layer,

7 CONCLUSIONS

The work of the MUSIC project has provided a coherent set of solutions which ease the task of developing context-aware, self-adaptive systems for mobile and ubiquitous computing scenarios. While most of the various aspects and pieces of building self-adaptive systems have been addressed before in other research projects, MUSIC stands out because it has looked at the complete picture and has delivered a comprehensive and coherent development framework, methodology, and execution platform.

The results of MUSIC have been adopted already by follow-on projects, and we expect that the development of self-adaptive applications and corresponding frameworks will grow in multiple directions in the future. Expanding the MUSIC framework for intelligent multimedia content adaptation is one challenging research alley, while other efforts will seek to ensure that the adaptations are as dependable as possible to bring autonomic computing to operation critical applications. The current version of the MUSIC middleware offers only basic support for security because the project intentionally focused on the described adaptation aspects. More work on the security features is needed. A very promising aspect of the MUSIC technology is its ability to support “systems of systems” with coordinated adaptation of sets of applications collaborating by discovering, providing and using services between them. This is based on state of-the-art service discovery and service level negotiation mechanisms. Thus, the MUSIC framework also provides a foundation for enabling applications to flexibly use services provided in the cloud, automatically adapting the usage of cloud-based services depending on quality of service and application context considerations. As this technology matures and the envisaged service market becomes a reality, MUSIC will not only be about ubiquitous computing for mobile users, but will represent a major step towards autonomic computing in general.

ACKNOWLEDGEMENTS

This work was partly funded by the European Commission through the project MUSIC (EU IST 035166), and by SINTEF and Telenor through their strategic research cooperation. The authors are indebted to the MUSIC project team which implemented the MUSIC framework and carried out the development and testing of the trial applications. Furthermore we thank the anonymous reviewers for their constructive comments and suggestions that helped to improve the quality of the paper substantially.

REFERENCES

MOFScript Home page, http://www.eclipse.org/gmt/mofscript/, Nov. 2011.
Andrieux, A., Czajkowski, K., Dan, A., Keahey, K., Ludwig, H., Kakata, T., Pruyne, J., Rofrano, J., Tuecke, S., Xu, M., 2007. Web Services Agreement Specification (WS-Agreement), Open Grid Forum Recommended Specification.
Ballagny, C., Hameurlain, N., Barbier, F., 2009. MOCAS: a State-Based Component Model for Self-Adaptation, SASO'09: The Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), San Francisco, pp. 517–524.
Brataas, G., Hallsteinsen, S., Rouvoy, R., Eliassen, F., 2007. Scalability of Decision Models for Dynamic Product Lines, SPLC 2007 Proceedins, the 2nd volume. Kinday Kagaku Sha Co. Ltd., Kyoto, pp. 23-32.
Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. A system of patterns - pattern-oriented software architecture. Wiley.
Capra, L., Emmerich, W., Mascolo, C., 2003. CARISMA: Context-Aware Reflective Middleware System for Mobile Applications. IEEE Transactions on Software Engineering 29, 929-945.
Cardellini, V., Casalicchio, E., Grassi, V., Iannucci, S., Lo Presti, F., Mirandola, R., 2011. MOSES: A Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems. IEEE Transactions on Software Engineering.
Charette, R.N., 2009. This car runs on code, IEEE Spectrum.
da Costa, C.A., Yamin, A.C., Geyer, C.F.R., 2008. Towards a General Software Infrastructure for Ubiquitous Computing. IEEE Pervasive Computing 7, 64-73.
Dey, A.K., 2001. Understanding and using context. Personal Ubiquitous Computing 5, 4-7.
Floch, J., et al., 2006. Using Architecture Models for Runtime Adaptability. IEEE Software 23.
Fraga, L., Hallsteinsen, S., Scholz, U., 2008. InstantSocial – Implementing a Distributed Mobile Multi-user Application with Adaptation Middleware, EASST Communications 11. EASST.
Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P., 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37, 46-+.
Geihs, K., Barone, P., Eliassen, F., Floch, J., Gjorven, E., Hallsteinsen, S., Horn, G., Khan, M.U., Mamelli, A., Papadopoulos, G.A., Papallis, N., Reichle, R., Stav, E., 2009a. A comprehensive solution for application-level adaptation. Software: Practice and Experience 39, 385-422.
Geihs, K., Reichle, R., Wagner, M., Khan, M.U., 2009b. Modeling of Context–Aware Self–Adaptive Applications in Ubiquitous and Service–Oriented Environments, Lecture Notes on Computer Science, Vol. 5525. Springer-Verlag.
Gomaa, H., Hussein, M., 2004, Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (Wicsa 2004), 79-88
Henricksen, K., Indulska, J., 2006. Developing context-aware pervasive computing applications: models and approach. Pervasive and Mobile Computing 2, 37–64.
ITU, 2009. Measuring the information society, The ICT Development Index. International Telecommunication Union.
Judd, G., Steenkiste, P., 2003. Providing contextual information to pervasive computing applications, Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications. IEEE Computer Society, Dallas-Fort Worth, p. 133.
Kakousis, K., Paspallis, N., Papadopoulos, G.A., 2010. A survey of software adaptation in mobile and ubiquitous computing. Enterp Inf Syst-Uk 4, 355-389.
Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer 36, 41-+.
Kuropka, D., Weske, M., 2008. Implementing a Semantic Service Provision Platform — Concepts and Experiences. Wirtschaftsinformatik Journal, 16–24.
Long, S., Kooper, R., Abowd, G.D., Atkeson, C.G., 1996. Rapid prototyping of mobile context-aware applications: the Cyberguide case study, Proceedings of the 2nd Annual International Conference on Mobile Computing and Networking. ACM, Rye, New York, pp. 97-107.
McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C., 2004. A taxonomy of compositional adap-tation, Technical Report Michigan State University.

Menasce, D., Dubey, V., 2007. Utility-based QoS Brokering in Service Oriented Architectures, Int. Conf. on Web Services (ICWS).
Paspallis, N., 2009. Middleware-based development of context-aware applications with reusable components. Ph. D. Thesis, University of Cyprus.
Paspallis, N., Rouvoy, R., Barone, P., Papadopoulos, G.A., Eliassen, F., Mamelli, A., 2008. A Pluggable and Reconfigurable Architecture for a Context-aware Enabling Middleware System, 10th International Symposium on Distributed Objects,. Springer Verlag LNCS 5331, Monterey, Mexico, pp. 553-570.
Poladian, V., Garlan, D., Shaw, M., Satyanarayanan, M., Schmerl, B., Sousa, J., 2007. Leveraging resource prediction for anticipatory dynamic configuration. First IEEE International Conference on Self-Adaptive and Self-Organizing Systems, 214-223.
Rouvoy, R., Eliassen, F., Floch, J., Hallsteinsen, S., Stav, E., 2008. Composing components and services using a planning-based adaptation middleware. Lect Notes Comput Sc 4954, 52-67.
Salehie, M., Tahvildari, L., 2009. Self-Adaptive Software: Landscape and Research Challenges. Acm T Auton Adap Sys 4.
Schmidt, D., Stal, M., Rohnert, H., Buschmann, F., 2000. Pattern-Oriented Software Architecture, Volume 2. Wiley, Chichester.
Scholz, U., Mehlhase, S., 2010. Co-ordinated Utility-Based Adaptation of Multiple Applications on Resource-Constrained Mobile Devices, in: Eliassen, F., Kapitza, R. (Eds.), DAIS 2010. Springer Amsterdam, pp. 198-211.
Sousa, J., Garlan, D., 2002. Aura: an architectural framework for user mobility in ubiquitous computing environments, in: Bosch, J., Gentleman, M., Hofmeister, C., Kuusela, J. (Eds.), 3rd Working IEEE/IFIP Conference on Software Architecture. Kluwer Academic Publishers, Montreal, pp. 29-43.
Wagner, M., Khan, M.U., Geihs, K., Lorenzo, J., Valla, M., Fra, C., Paspallis, N., Papadopoulos, G.A., 2008. A Comprehensive Context Modeling Framework for Pervasive Computing Systems, 8th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS). Springer Verlag LNCS 5053, Oslo, pp. 281-295.
Want, R., Hopper, A., Falcão, V., Gibbons, J., 1992. The active badge location system. ACM Transactions on Information Systems 10, 91-102.
Weiser, M., 1991. The computer for the 21st century, Scientific American. Scientific American.