SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:385–422
Published online 15 September 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.900

A comprehensive solution for application-level adaptation

K. Geihs[1,*,†], P. Barone[2], F. Eliassen[3,7], J. Floch[4], R. Fricke[5], E. Gjorven[3], S. Hallsteinsen[4], G. Horn[4], M. U. Khan[1], A. Mamelli[2], G. A. Papadopoulos[6], N. Paspallis[6], R. Reichle[1] and E. Stav[4]

[1] *University of Kassel, D-34121 Kassel, Germany*
[2] *HP Italy Innovation Center, I-20063 Cernusco sul Naviglio (Milano), Italy*
[3] *Simula Research Laboratory, NO-1325 Lysaker, Norway*
[4] *SINTEF ICT, NO-7465 Trondheim, Norway*
[5] *Condat AG, D-10559 Berlin, Germany*
[6] *University of Cyprus, CY-1678 Nicosia, Cyprus*
[7] *University of Oslo, NO-0316 Oslo, Norway*

SUMMARY

Driven by the emergence of mobile and pervasive computing there is a growing demand for context-aware software systems that can dynamically adapt to their run-time environment. We present the results of project MADAM that has delivered a comprehensive solution for the development and operation of context-aware, self-adaptive applications. The main contributions of MADAM are (a) a sophisticated middleware that supports the dynamic adaptation of component-based applications, and (b) an innovative model-driven development methodology that is based on abstract adaptation models and corresponding model-to-code transformations. MADAM has demonstrated the viability of a general, integrated approach to application-level adaptation. We discuss our experiences with two real-world case studies that were built using the MADAM approach. Copyright © 2008 John Wiley & Sons, Ltd.

Received 5 December 2007; Revised 7 July 2008; Accepted 8 July 2008

KEY WORDS: adaptive software; component framework; compositional adaptation; context awareness; model-driven development; utility function

*Correspondence to: K. Geihs, University of Kassel, D-34121 Kassel, Germany.
†E-mail: geihs@vs.uni-kassel.de

Contract/grant sponsor: EC; contract/grant number: FP6 IST 4159

1. INTRODUCTION AND MOTIVATION

Mobile and pervasive computing introduces a growing demand for software systems that are able to adapt to dynamically changing environments. Generally, we differentiate between static and dynamic adaptations: Static adaptation relates to the redesign and reconfiguration of application architectures and components at design time when functional and non-functional requirements change. Dynamic adaptation happens at application run-time due to changing resource and context conditions. For example, applications may want to react dynamically to fluctuations in network connectivity, battery capacity, appearance of new devices and services, or to a change of user preferences.

In this paper we focus on dynamic adaptation of context-aware applications on mobile computing devices. We present the results of MADAM, a joint European research project with eight partners from industry and academia from all over Europe [1]. Mobile computing has been chosen as our main application scenario as dynamic changes in the execution context and the need for application adaptation will be evident, especially in such scenarios where users move from one location to another, e.g. from home to office to customer premises.

Clearly, MADAM has not been the first project to tackle dynamically adaptive software systems. It is based on, incorporates, and extends earlier research work. However, MADAM stands out as it provides a truly comprehensive solution that addresses adaptation from both the theoretical and the practical perspective and solves all of the following challenges:

- Adaptation happens seamlessly and without user intervention in reaction to context changes.
- Applications may exploit any kind of context dependencies as long as there is appropriate hardware and software available in the computing environment to provide these context data.
- Context-awareness and adaptivity of applications are treated as a separate concern in application design.
- A general component model and middleware infrastructure support many adaptation styles, e.g. local and distributed adaptation, parameter and compositional adaptation, anticipated and unanticipated adaptation.
- A model-driven development approach comprising adaptation models and corresponding transformations facilitates the development of self-adaptive applications and the reuse of adaptation artefacts.
- Real applications from industry partners are used to evaluate the approach.

Although many different aspects of adaptive software systems have been explored before (see Section 6 and the special issue [2]), in MADAM we have taken a generalized research approach that integrates the majority of those aspects. Thus, the main contributions of this paper are:

- We demonstrate the viability of a general, integrated approach to application-level adaptation for context-aware applications.
- We present a flexible system infrastructure that enables dynamic adaptation in many ways; it can be realized on top of different component platforms such as Java and .NET.
- We introduce a new general-purpose, model-based development methodology for adaptive applications that builds on UML and Eclipse.

In this paper we present a retrospective on the overall approach, the main achievements, and the general conclusions of project MADAM. Owing to space limitations and the large amount of information, we cannot go into full detail in some sections. In these cases we refer the reader to previous publications where more details can be found. The structure of the paper is as follows. Section 2 discusses adaptation requirements and explains how these requirements are translated into a novel generalized adaptation approach. Section 3 presents a component-based computing infrastructure that supports dynamic adaptation at the application level. In Section 4 we address the question of how to assist the developer in the design and implementation of general-purpose adaptive applications. A model-driven development methodology is presented that considerably eases the development task. In Section 5 we describe two real-world case studies that were built using the MADAM approach; we present performance measurement results, and we discuss our experiences and lessons learned. Related works are analyzed in Section 6. Finally, Section 7 presents conclusions and outlook for future work.

2. ADAPTATION REQUIREMENTS AND APPROACH

The increasing proliferation of mobile devices has affected the expectations of their users. Mobile users now expect their devices to automatically and intelligently adapt to their ever-changing context. However, such an adaptive behaviour can be quite difficult to develop, especially when the nature of the adaptations is complex and difficult to describe. Adding to the complexity is the fact that different sets of context information are available on different devices and also that different users have varying preferences concerning the desired adaptations.

The work of the MADAM project has endeavoured at providing solutions that ease the task of developing context-aware adaptive systems for mobile computing. As a first step in the project, we sketched a number of scenarios that were then used to extract a set of requirements in the form of reference adaptations. This list of requirements has guided us through the design of the MADAM middleware and methodology.

2.1. Requirements

The MADAM case studies were based on both real commercial products and on hypothetical scenarios. In the following we describe a simplified scenario, which is derived from one of the case studies: the SatMotion application. This scenario is revisited in the subsequent sections. For the complete scenario and the case study, the interested reader is referred to the project deliverables D1.2 [3], D5.3 [4], and D6.2 [5].

2.1.1. Application scenario

Consider a maintenance worker who uses an application running on her mobile device (e.g. a PDA) for her on-site work. Because of the nature of her tasks, the worker might or might not be able to visually interact with the mobile device. Consequently, the application is designed to offer multiple modalities for user interaction, i.e. visual and audio interaction. The visual interaction displays text messages (e.g. in popup windows) while the audio interaction uses the PDA's speakers and microphone for interaction with the user. Depending on the user's context, the middleware automatically selects the available modes that are to be chosen. Additionally, the application is capable of running a speech-to-text service both locally on the user device and remotely on another node. When the audio interaction is used, the adaptation reasoning mechanism automatically decides whether hosting the text-to-speech engine locally is more beneficial than hosting it remotely on a predefined company server, and vice versa.

For example, consider the case where the worker is in her office, interacting with the PDA in order to prepare herself for an on-site visit. In this case, the application has the full attention of the user, and consequently the visual interaction is selected as it is more responsive (e.g. faster interaction) as well as more resource efficient (i.e. requires less memory and CPU, and uses no networking). However, when the worker moves to the site and starts working with the equipment, the application switches to audio interaction, thus releasing the worker from the burden of having to visually interact with the application. Finally, when the network is sufficiently fast and cheap, and the resources are low (e.g. because the PDA starts other applications as well), the application switches to the audio mode where the speech-to-text component is hosted on a remote predefined server.

2.1.2. Reference requirements

Based on scenarios such as the one above, we concluded a set of reference requirements in the form of possible adaptations that the MADAM solution should support:

User interface delegation: Transfer of (part of) the UI functionality to another device. For example, while driving a car a mobile user may have the option of connecting the mobile device to the onboard car computer in order to enable a hands-free interaction mode with the mobile device.

User interface presentation: Tune the UI such that it is optimized for the current context conditions. For example, based on the ambient light conditions of its location, a mobile device could adjust its screen brightness.

User and application session redeployment: Redeployment of components, and/or applications to different devices, to accommodate for more efficient use. For example, in the office parts of the user application can be redeployed from her handheld to her desktop PC.

Functional richness: Extension of the functionality of an application by providing access to new software or hardware components. For example, an application that controls the home cinema in a living room could extend its functionality if a new hardware plug-in was added for controlling the lights of the room.

Data richness: React to the dynamically changing quality of a data stream. For example, a video conference application might want to adapt to different network characteristics, such as available bandwidth and latency by adjusting the resolution and the colour-depth of the video.

Network availability: Adapt to different mobile network technologies, e.g. GPRS, WLAN, Bluetooth, etc. For example, a mobile device can switch from a WLAN connection when the user is in the office to a GPRS connection when the user is leaving the building.

Security: Adjust the security mode based on the execution context. For example, communication via a GPRS communication link requires strong encryption, but when the user returns to the office and connects to the local WLAN, encryption may be turned off, thus saving resources on the device.

Software mode: Switch between different application-specific modes of operations. For example, a video streaming system may select different compression algorithms depending on data rates and communication quality.
Data replication and synchronization: Automate the replication and synchronization of data both in space and in time. For example, switch from local data to a server data base when the communication link is fast and cheap, or defer the downloading of large documents until the network is sufficiently fast or cheap.
Changing user preferences: Cater for changing user needs and preferences during the application lifecycle. Thus, the user needs and preferences should be considered as part of the context. For example, the user may want to specify different application priorities for different situations, such as work and leisure.

Although these adaptation types appear to be sufficiently simple at an individual level, it is argued that programming a system to achieve several of these adaptations in parallel can be quite challenging. Additionally, it is rather obvious that the development process for adaptive applications involves a number of repeated steps, such as defining how to collect the context information and how to decide about the adaptations. Hence, there is a need for a more methodical and automated approach in order to rid the developers of complex and repetitive work.

2.1.3. Adaptation constraints

Although the above requirements represent items on our wish list for a general adaptation framework, we have to take into account certain inherent constraints and limitations. The most important ones are related to the resource constraints on mobile devices and to the distributed nature of applications.

Typically, mobile computing devices are constrained in terms of processing power, storage capacity, and communication capabilities. Furthermore, battery power is limited. Thus, resource consumption and scalability of adaptation infrastructures are major concerns. Generally, the adaptation overhead is a crucial point in the design of such systems. In order to achieve scalability, the number of variants that are evaluated at run-time in reaction to a context change must be kept as low as possible by trying to eliminate configurations upfront that are not feasible in the given situation. In Section 4 we will present the solution that was adopted by MADAM.

Distribution is inherent to mobile applications and an important aspect in the design of adaptation frameworks. Additionally, distribution is useful as a means of alleviating the resource constraints of mobile computing devices. Hence, we must acknowledge the fact that applications may be distributed over client and server nodes, and that application components running on clients may access communication, processing, and storage services provided on a server. Consequently, application adaptation decisions may involve local and remote contexts and application components, i.e. MADAM provides distributed context and adaptation management.

2.2. Adaptation approach

From the discussions so far, it is obvious that multiple context elements may be relevant in a mobile setting and several forms of adaptation may contribute to enhance or maintain the service quality when the context changes. This section deduces the design of a general architecture for self-adapting systems and explains the adaptation rationale.

Three main middleware functionalities are required for the adaptation of applications:

- Monitor the context, detect context changes, and reason about the relevance of these changes.
- Make decisions about what adaptation to perform.
- Implement the adaptation choices, i.e. reconfigure the application(s).

First, let us consider context monitoring. We observe that multiple context elements need to be taken into account. Further, these span from elementary elements, such as network cost, to more complex aggregated or derived elements, such as predicted location. We expect that the set of relevant context elements and the sources producing them will evolve over time in the same way as applications do. Our experience with defining mobile scenarios is that often new context elements are introduced gradually as the problem to be solved becomes better understood or new application functionalities are added. All these observations lead to the following architectural implications:

(i) Context monitoring should be kept separate from the application and realized through reusable context middleware. The context middleware should be extensible and support the addition of new context elements and new forms of reasoning.

Beyond reusability and extensibility of context components, the separation of concerns enforced by these implications should simplify the development of adaptive applications. Context management becomes the responsibility of components outside the applications. The context middleware can be developed, modified, and extended independently from the applications.

Second, concerning adaptation reasoning, we observe multiple relations between context and adaptation mechanisms as well as interfering adaptation effects. Thus, context elements cannot be considered separately when reasoning on adaptation. For example, the audio capabilities should be considered together with the user activity before selecting the UI modality (i.e. voice or text-based UI). Also the implementation of an adaptation may have effect on the context. For example, the selection of the UI modality has an impact on the consumption of resources of the handheld device.

The more context elements we introduce, the more relations we need to deal with. From our experience with the generalization of scenarios we know that it is rather difficult to capture all these relations [6]. We also expect that new relations will be introduced as applications and context elements evolve. In [7] two kinds of approaches have been examined for self-adapting, context-aware applications: internal approaches where context-management and adaptation are realized as part of the application using programming language features, and external approaches where these mechanisms are realized by an application-independent middleware. The main drawback of internal approaches is the complexity introduced by intertwining adaptation and application behaviours. In addition, they poorly support the evolution of the software. These drawbacks make internal approaches inappropriate in the context of mobile services, and thus we chose externalization of adaptation mechanisms:

(ii) Adaptation mechanisms should be realized externally to the application.

External approaches require adaptation policies to be described separately from the applications. These policies are used by the middleware to reason and decide about adaptation. Three main approaches have been proposed for the description of such policies:

- Situation-action approaches specify exactly what to do in certain situations [8,9]. Action-based specifications are undoubtedly the most popular and are used in different domains related to networks and distributed systems such as computer networks, active databases, and expert systems.

- Goal-oriented approaches represent a higher-level form of behavioural specification that establishes performance objectives, leaving the system or the middleware to determine the actions required to achieve those objectives [10]. Goal specifications capture the relations between context and adaptation mechanisms in a concise way.
- Utility-based approaches extend goal-oriented approaches [11,12]. Utility functions ascribe a real-value scalar desirability to system states (i.e. in our case, a state is an application variant). The middleware computes the utilities of variants and selects the variant with the highest utility.

Situation-action approaches require the explicit description of each situation. The more the context elements, the more the conditions and rules need to be specified. Furthermore, these approaches use simple rules that fail to catch some dependencies between adaptation and context. They are therefore not appropriate for the purpose of our work. Using goal-oriented approaches, it is possible to express policies at a higher level. However, goal-oriented approaches also have major drawbacks. They fail to catch dependencies between adaptations and goals, and conflicts between goals. For example, a goal towards the optimization of CPU usage can be achieved through the suspension of lower priority applications, but such suspension would have impact on a goal towards high service availability. In addition, they do not provide any mechanism to compare adaptation actions when several actions can be applied to achieve a goal. Utility functions express the rationale for adaptation decisions in a precise way, and are therefore more appropriate than goal policies when adaptation triggers and effects interfere, or when goals are in conflict. In our case, utility functions can be specified to express the dependencies between the properties of the system variants, the context and the user needs, and thus we chose a utility-based approach to tackle the complexity of adaptation reasoning:

(iii) Adaptation policies should be expressed using utility functions.

Although utility-based approaches allow precise decision making, they require developers to specify in detail the properties of the application variants. The utility function computes the utility of an application variant as a function of the predicted properties of the different variants and the current context. The aim is to select the application variant that maximizes the utility of the application while satisfying the resource constraints provided by the underlying environment. It has turned out that defining the utility function for complex applications might be a difficult task. Methodological help and appropriate tools are needed in order to support this. For example, when developing MADAM applications it often proved that the utility function of an application could appropriately be 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. In our experience, adaptation policies for multiple applications can similarly be expressed as a weighted sum of individual application utilities where the weights express application priorities of the user. When resources are scarce this may lead to adaptations where low priority applications will have to be suspended in order to maximize overall user satisfaction.

(iv) Adaptable applications should be built on compositional variability combined with support for component parameterization.

Finally, we need to decide what kind of adaptation mechanism should be employed. Two popular adaptation techniques are parameter adaptation and compositional adaptation [13,14]. Parameter adaptation supports fine tuning of applications through the modification of program variables and deployment parameters, whereas compositional adaptation allows the modification of the application component structure and the replacement of components. Parameterization is an intuitive and effective way to implement variability, but it is less powerful than compositional adaptation and may also raise scalability concerns if the possible parameter value ranges are very large. Thus, our focus is on compositional adaptation. Note that it is an interesting open question whether the two adaptation mechanisms can cover all possible scenarios of dynamic application adaptation. Our experience suggests that this is indeed the case.

In summary, MADAM has adopted an external adaptation approach where adaptation capabilities of applications are realized by an application-independent middleware. Adaptation is based on compositional adaptation with added support for parameter adaptation. We follow an architecture-centric approach and employ architecture models at runtime to allow generic middleware components to reason about and control adaptation.

Figure 1 summarizes the main ideas of the MADAM adaptation approach. The middleware realizes three main adaptation-related functions: context management, adaptation management, and configuration management. To fulfil these functions, the adaptation middleware needs to know the application structure and constraints, as well as the various context and resource dependencies. This implies that the middleware works at runtime on an architectural model of the application that specifies all adaptation variants and constraints. Note that this design enables unanticipated adaptations where new components or services are evaluated at runtime that were not available at application design time.

The MADAM approach introduces a new major task for the developer: the building of application architecture models needed by the adaptation middleware to perform adaptation reasoning. Therefore, we have developed a software development methodology based on the model-driven architecture (MDA) paradigm. The application developer captures the application variability in platform-independent adaptation models. The adaptation models are transformed by transformation tools into a representation (i.e. Java code in our case), that the middleware can access and leverage at runtime for adaptation management. This methodology is explained in detail in Section 4.

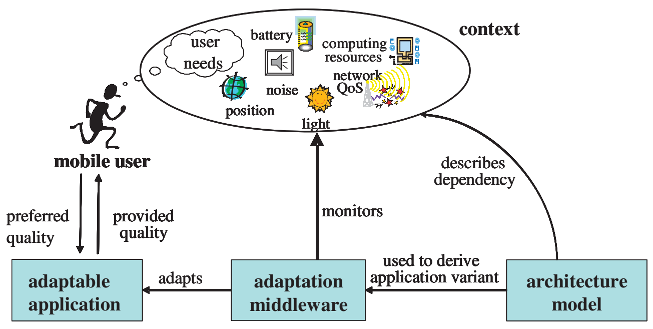


Figure 1. MADAM approach at a glance.

3. ADAPTATION MIDDLEWARE

This section presents the design of the component model and computing infrastructure that satisfy the above requirements.

3.1. Adaptation based on component frameworks

MADAM uses component frameworks to design applications that are capable of being adapted by reconfiguration. The component framework defines the required roles and interfaces of components that are plugged into the framework and regulates the interaction between components.

In MADAM, a component framework describes a composition of component types. These types represent variation points at which various component implementations can be plugged in. Variability is achieved through the plug-in of component implementations whose externally observable behaviour conforms to the component type. A component plugged into a component framework may be an atomic component, or a composite component built as a component framework itself. In this way, an application can be assembled from a recursive structure of component frameworks.

Figure 2 shows how components relate to applications and how variability is achieved. An application is considered as a component type that can have different realizations. The realization details of a component type are described using plans. Components can be atomic as well as composite. Consequently, there are two types of plans: blueprint and composition plans. A blueprint plan describes an atomic component and contains just a reference to the class or the data structure that realizes the component. The composition plan describes the internal structure of a composite component by specifying the involved component types and the connections between them. Variation is obtained by describing a set of possible alternative realizations of a component type using plans.

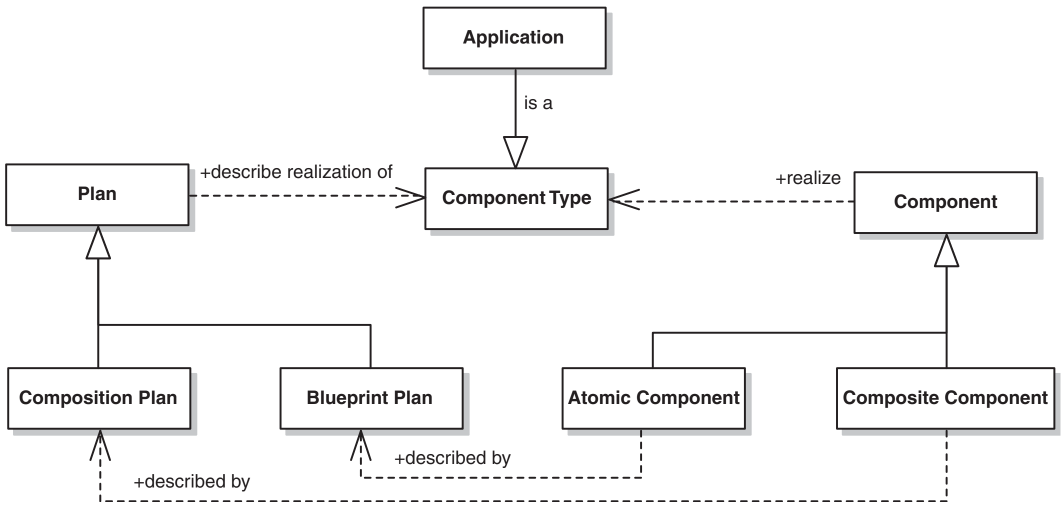


Figure 2. Creating application variants.

MADAM also supports variability through component parameterization [15]. Component behaviour may be adapted by setting component parameters to appropriate values. This supports fine-grained adaptation and may be used to fine-tune component behaviour such as tuning the encoding of a video stream to fit the available bandwidth. See Section 4 for details.

To discriminate between alternative component implementations MADAM components are annotated with properties. Properties are used to qualify the services offered or needed by components. For example, properties can describe that a user interface component implementation supports a hands-free mode. Properties can also specify requirements to system resources such as required memory space or network bandwidth. Properties are functionally dependent on context elements. In this way we are able to represent the dependencies between component implementations and context.

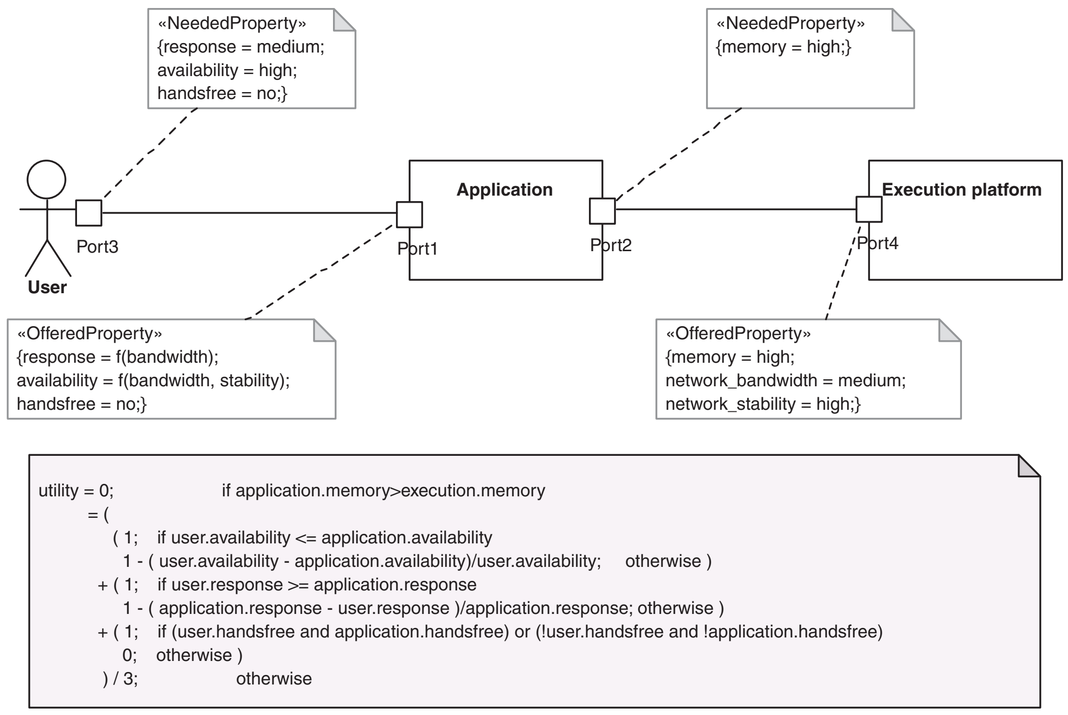
As components may provide several services and collaborate with multiple components, we need to distinguish between service quality on each collaboration. To do so, we use the concept of port. Components have ports through which they interact with each other. Thus, a port represents the capability of a component to participate in a specific kind of interaction or service. Properties are attached to ports. Services needed from the underlying platform such as memory are represented by implicit component ports, and the platform-related properties attached to these ports. Similarly, services provided to the user are represented by implicit component ports. It is the component type that defines that set of properties and their associations with ports.

Associated with implementations, property predictor functions are used to predict the properties of the implementations in a given context. Property predictors are needed because adaptation (i.e. reconfiguration) of applications will in general have influence on the state of the resources of the computing platform. In other words, we need to take into account that adaptation will change the system context. Property predictors can be implemented in various ways including as constants, for instance, to express that a specific component implementation requires more than 500 kB memory, or as functional expressions of other properties of the component itself, of collaborating components, or of inner components in the case of composite components. For instance, the response time offered by a component may depend on the network bandwidth available to that component, or the memory needed by a component may depend on the memory needed by inner components.

The concepts of property and port are illustrated in the upper part of the example in Figure 3. The user interacts with the application in a specific context, as defined by the property values shown.

The lower part of Figure 3 displays a simple example for a utility function. Utility represents a measure of how well an application configuration fits a given context. The utility of an application variant is computed as a function of the properties of the ports representing the interaction between the application and its context. In the example, the utility depends on whether the offered availability and the offered response time satisfy the values required by the user and whether the required and offered properties ‘hands-free’ have identical values. Thus, the utility function takes into account properties of the application and the current execution environment as well as user preferences. The utility function is specified by the application developer. In the current MADAM prototype, utility functions compute the weighted sum of the differences between offered and needed properties where the weights reflect the priorities of user needs. Within this model the goal of adaptation management can be formulated as ensuring that at any time the running system variant is the one with the highest utility that does not violate the resource constraints imposed by the environment.

Distributed applications are designed using connectors that abstract communication protocols between remote components. Figure 4 shows an example of applying the MADAM component



```
utility = 0;                    if application.memory>execution.memory
        = (
           ( 1;   if user.availability <= application.availability
             1 - ( user.availability - application.availability)/user.availability;    otherwise )
         + ( 1;   if user.response >= application.response
             1 - ( application.response - user.response )/application.response; otherwise )
         + ( 1;   if (user.handsfree and application.handsfree) or (!user.handsfree and !application.handsfree)
             0;   otherwise )
          ) / 3;                otherwise
```

Figure 3. A context-aware application and its utility function.

model. It illustrates the component model of a simple client–server application. The *STappImpl* component is a composite component describing the component framework implementing the top-level application type *STapp* (service technician application). Each role defined by this framework, *UI* (*user interface*), *Ctrl* (*control*) and *Db* (*database*) is mapped to alternative component implementations at run-time. *Db* has two implementations that are also built as component frameworks.

In practice, as the service technician progresses through her daily tasks, the context changes either because she moves on to a different site, or because the environment changes (i.e. different time of day, different light conditions, weaker wireless network signals, etc.). As a consequence, the possible configurations are evaluated using the utility function, and the application is appropriately adapted. For instance, the UI role is dynamically resolved to be implemented by either the visual (normal) or the audio (hands-free) UI, depending on the user context (are her hands busy driving or installing an antenna, or not).

3.2. Middleware architecture

Figure 5 illustrates the MADAM middleware architecture and its most important interfaces. As explained before in Section 2, the middleware performs three main adaptation-related functions: context management, adaptation management, and configuration management. We will present these

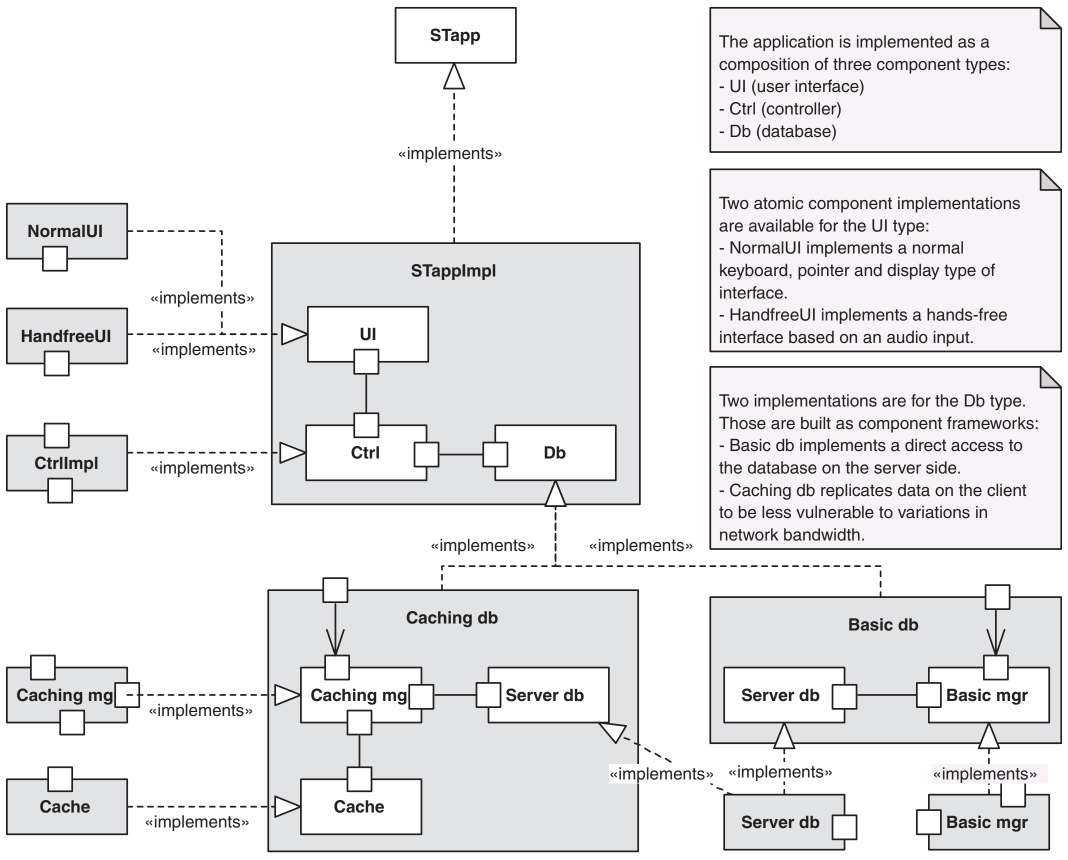

STapp
«implements»
The application is implemented as a composition of three component types:
- UI (user interface)
- Ctrl (controller)
- Db (database)
Two atomic component implementations are available for the UI type:
- NormalUI implements a normal keyboard, pointer and display type of interface.
- HandfreeUI implements a hands-free interface based on an audio input.
Two implementations are for the Db type. Those are built as component frameworks:
- Basic db implements a direct access to the database on the server side.
- Caching db replicates data on the client to be less vulnerable to variations in network bandwidth.
NormalUI
HandfreeUI
CtrlImpl
STappImpl
UI
Ctrl
Db
«implements»
Caching db
Basic db
Caching mg
Server db
Cache
Basic mgr
Server db
Basic mgr

Figure 4. Component framework architecture for the service technician application.

middleware services in more detail in the following sections. In addition, two more components play a vital role inside the middleware, i.e. the Resource Manager and the Middleware Core.

The Resource Manager provides information on the state of the resources. It can be queried for the state of a resource in a pull mode, or components can subscribe to resource state change notifications in a push mode. Furthermore, it handles locking and reservation requests, and collaborates with remote Resource Managers in case of distributed adaptation decisions. The Resource Manager is primarily accessed by the Context Manager for monitoring context related to the device resources (e.g. memory, CPU, networking) and by the Adaptation Manager during the adaptation process.

The MADAM Middleware Core provides platform-independent interfaces for the management of applications, plans, components, and component instances. It is used to instantiate, initialize, and remove component instances, to connect and disconnect instances, to create remote bindings to remote components, to install, uninstall, and locate plans, and more. The Core and the Adaptation Manager collaborate in order to launch and manage applications. When an application launch request is received by the Core, it is forwarded to the Adaptation Manager, which in turn uses Core interfaces to locate, instantiate, and bind the components participating in the application.

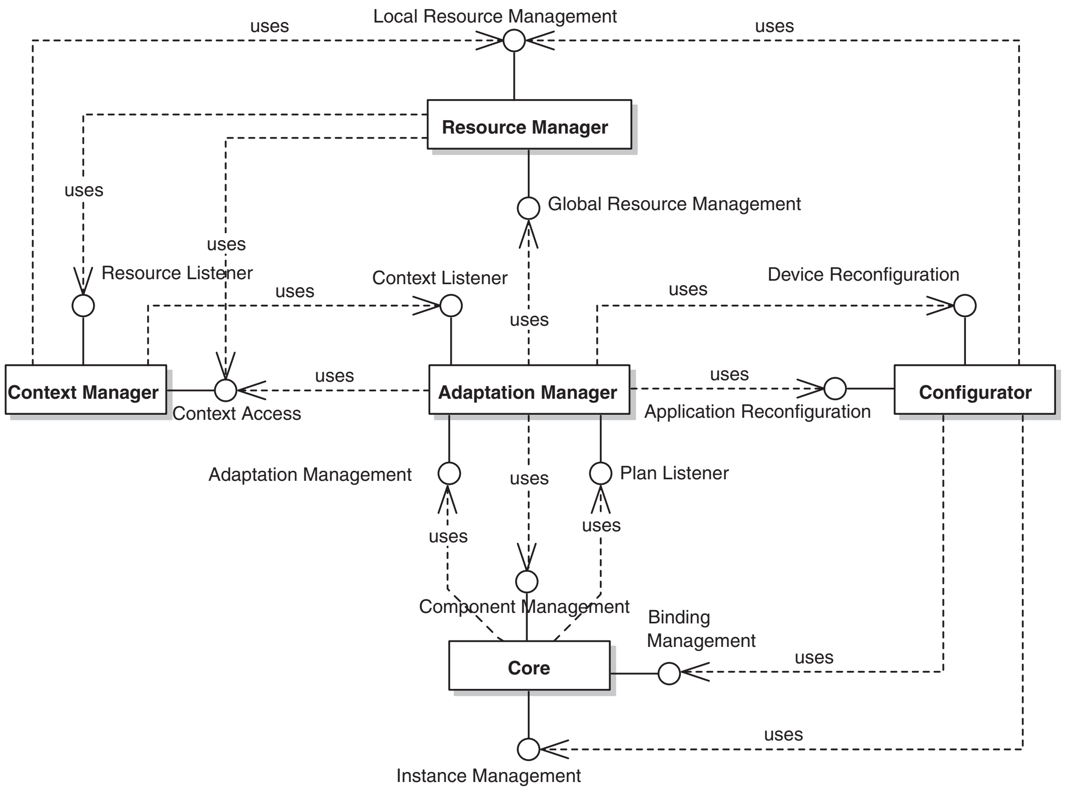


Figure 5. Middleware architecture.

The MADAM middleware supports distributed adaptation, i.e. adaptation that involves activities on several nodes. All nodes participating in a distributed adaptation need to run the MADAM middleware. The MADAM nodes are connected by remote bindings. One node acts as master (mostly the mobile device carried by the user, i.e. the node on which the core components of the application run), the others as slaves. The master makes the adaptation decisions and potentially delegates configuration operations to the slave nodes. Thus, the distributed adaptation management protocol is based on centralized adaptation reasoning and potentially distributed re-configuration. Clearly, centralized decision making is subject to scalability problems. Note, however that in typical mobile computing scenarios a typical mobile device, such as a PDA, runs just a few fairly simple concurrent applications and maintains just a few connections to other devices at a time. Hence, our design approach caters well to the given requirements and constraints. See also Section 5.3 for performance measurements.

3.3. Context management

One of the main goals of the MADAM approach was to increase the reusability and to ease the development of context-aware, adaptive applications. In this respect, the context middleware was defined in a way that enables the developers to treat the collection and management of context information as an independent, cross-cutting concern.

3.3.1. Context model

By context, we refer to '*any information that can be used to characterize the situation of an entity*; [*where*] *an entity is a person*, *place*, *or object that is considered relevant to the interaction between a user and an application*, *including the user and the application themselves*' [16]. More precisely, our definition of application context assumes that an application has only one context that consists of a number of context elements whose values may change during run-time. Application context can be subdivided into three basic categories: user, computing, and environmental context [17]. User context reflects the user's preferences and profile, computing context reflects the state of the system resources (e.g. memory, battery capacity, bandwidth), and environmental context is determined by available sensors that provide, for example, measurements of the current location, speed, or temperature. Although many works in the literature have focused on certain types of context data, such as location-related context, in our work we have opted for a general approach, allowing the developers to specify, model, and use any type of context data. In this respect, a simple and extensible model was defined, allowing the developers to specify arbitrary types of context entities and context values.

The basic concept of this model is the *ContextEntity*. *ContextEntities* can include additional (child) *ContextEntities*, thus forming a tree-like structure. The leaves of this tree are the context values, which include numerical or string-based representations for the abstracted context data. This model also allows for the specification of a limited and predefined set of metadata, such as a time-stamp, the ID of the source that provided the context data, the probability that this data is correct, and the user rating when it is available. These metadata are primarily aimed at the internal context management mechanism, but they can also be accessed by context clients. More detailed descriptions of the MADAM context model are available in [18].

3.3.2. Context Manager

The basic philosophy of the MADAM context architecture revolves around two points: First, context clients have specific context needs, which they publish to the *ContextManager*. Second, context providers offer specific types of context data, which are also published to the *ContextManager*. For this purpose, the *ContextManager* provides a context repository that acts as a central hub for the routing of context information among sensors, reasoners, and consumers, as is illustrated in Figure 6.

The lifecycle of context data consists of several phases: collection of context data (by sensors), local and remote distribution, caching and storing, processing, and eventually consumption of context data. Sensors are used to populate the context repository with context data. Context clients, such as external applications and the MADAM middleware register with the *ContextManager* for any of the two types of context change notifications: synchronous and asynchronous. The former provides a request/reply type of direct access to the context repository, while the latter uses the publish/subscribe pattern in order to enable event-based asynchronous notification.

For the purposes of the context middleware an extensive plug-in mechanism was developed, allowing the developers to dynamically add special mechanisms that feed the main *ContextManager*

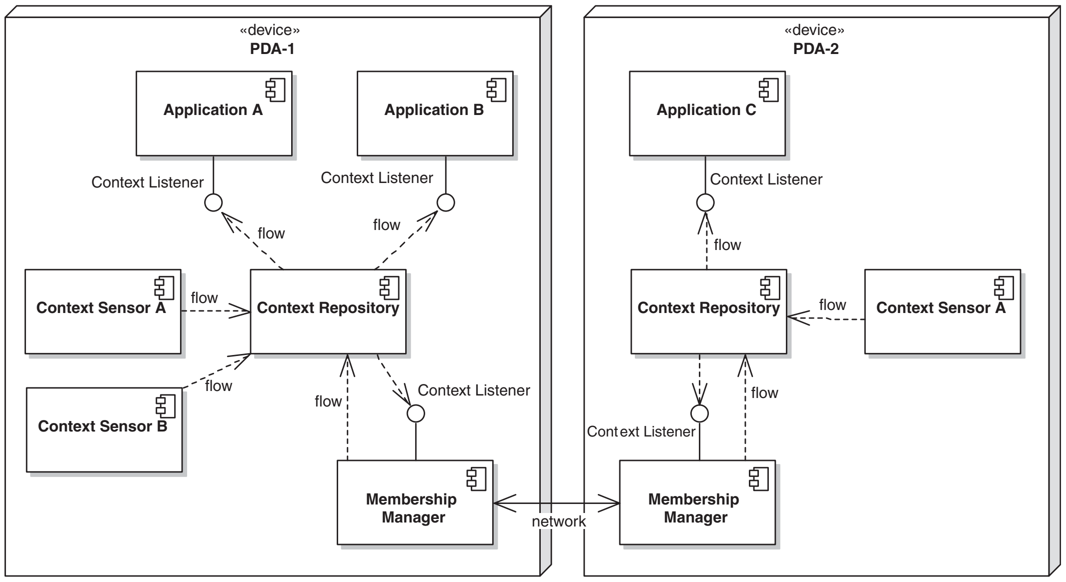


Figure 6. Local and distributed context management.

with context data. Thus, developers can dynamically add new functionality to the middleware, e.g. by packaging relevant context sensors along with the applications that need them, and the middleware can automatically start and stop context sensors depending on the context needs of the currently deployed applications. The latter enables the system to better utilize the resources, which are particularly scarce in the case of mobile devices.

Besides pluggable context sensors, the MADAM architecture also defines pluggable context reasoners. The latter are specializations of sensors providing the additional capability of accessing context data from the repository with the purpose of processing it and deriving more advanced context information.

Once collected, the context data is preserved locally in a repository. The amount of data that is stored in the repository is configurable. The default is to only store the latest value of each context type, but this can be changed to allow for storing historical values. The latter is particularly useful in the case of context predictors, i.e. specialized context reasoners that try to infer the value of a context element in the future, with a given probability, by assessing historical context values.

As MADAM primarily aims at distributed applications, one of the most advanced features of the *ContextManager* is distributed context management across multiple devices over either infrastructure-based or *ad hoc* networks. This is achieved in a way that is transparent to both users and developers. For this purpose, a specialized component is defined: the *MembershipManager*.

The *MembershipManager* achieves two main tasks: management of group memberships and communication of context information across remote instances of the MADAM middleware. Groups are dynamically formed and contain instances of the middleware that are within reach of each other and willing to share their context information. The latter allows any member of the group to unicast or multicast any type of context information to other members of the group.

The basic assumption of this approach is that the context nodes publish their context requirements, and at the same time they collect and maintain the context needs of the other nodes. The distribution mechanism exploits this, and extends it across distributed instances of the middleware. In this respect, it is possible for a context client that resides on one device, to request a context entity of a type T, and receive it through a sensor, which is attached to a different device within the same membership group. To enable this type of functionality, the membership managers implement a specialized protocol that broadcasts messages encoding the (locally unsatisfied) context needs of the applications running on the devices. As the membership manager uses periodic broadcasts for maintaining the membership groups, the context needs are encoded and piggy-packed to the normal membership heartbeats for efficiency reasons. As a result, the context space of each participating node is extended to include context information available in neighbouring, directly accessible nodes.

Of course the distribution of context information has several implications, most notably on security and privacy aspects. As context data may include sensitive, user-related information, the context model is appropriately extended so that it classifies context types in different categories, according to their applicability or desirability for distribution. For more details on the modelling aspects, as well as on the local and distributed context management architectures the interested reader is referred to [19].

3.4. Adaptation management

Each component may have a number of different realizations that provide the same basic functionality, but differ in their extra-functional characteristics like resource requirements and context dependencies, as described in Section 3.1. The MADAM component model describes how variability can be achieved by plugging in different component implementations whose externally observable behaviour conforms to the component type, but whose properties, which are described using the property model, are different.

In the MADAM middleware the *AdaptationManager* is responsible for reasoning on the impact of context changes on the applications, and for adapting the set of running applications by planning and selecting the application variants and the device configuration that best fit the current context. It consists of two subcomponents, the *AdaptationCoordinator* and the *BuilderandPlanner*. The *AdaptationCoordinator* is responsible for coordinating the actions to take place when a context change occurs and for selecting variants for the set of concurrent applications. In order to select the fittest variant, the *AdaptationCoordinator* needs to determine all the possible variants for each application, and to evaluate them. The *BuilderandPlanner* is responsible for dynamically building descriptions of alternative application variants represented as configuration templates. Thus, for each of the possible application variants and also for each of its possible deployments a configuration template is created. During the evaluation of a configuration template, first the *AdaptationCoordinator* checks whether the resource requirements of the application variant can be satisfied. If a deployment involves several nodes, it contacts not only the local *ResourceManager* but also the remote *ResourceManagers*. Only if all resource requirements can be satisfied, the utility of the configuration template is evaluated. In the case of a distributed application not only local context information but also context information from the other nodes is used to calculate the utility of the configuration template. The *AdaptationCoordinator* maintains a *BuilderandPlanner* for each concurrent application, and uses the configuration templates to assess the utility of the variants in the current context and select the 'best' variants.

The *AdaptationManager* manages applications throughout their life cycle:

- *Application launch*: When a client starts a new application, the *AdaptationManager* builds a model of the adaptive application by retrieving the plans that describe how the application's component types can be realized, recursively traversing the composition structure. In order to be able to manage the adaptation of the application, it registers to receive notification about context changes relevant for that application. When the *AdaptationManager* has determined the best set of application variants, the *Configurator* (explained below) is requested to instantiate this application configuration, and to configure the device using the device settings determined to best match the current context and the running set of applications. For distributed applications, the corresponding *Configurators* on the other nodes are also requested to instantiate the required application components.
- *Application run-time adaptation*: Adaptation is triggered by context information received by the *AdaptationManager* from the *ContextManager*. Depending on the context information, the *AdaptationManager* may decide to replace the running application variant with an alternative variant, and invoke the *Configurator* to perform reconfiguration of the applications and the device.
- *Application shutdown*: On application shutdown, the *AdaptationManager* performs the necessary cleanup for the terminating application and find the best configuration for the remaining set of running applications.

3.5. Configuration management

The responsibility of the MADAM configuration management is to orchestrate the changes required to safely reconfigure from the current set of running application variants to the new set of application variants selected by the adaptation management. This includes starting and stopping of applications and their components, configuring their connections and parameters, and changing device settings.

The main component handling the configuration management in MADAM is the *Configurator*. It is responsible for coordinating the initial configuration and later reconfigurations of the application components and the device. The *AdaptationManager* and the *Configurator* are tightly coupled as they operate on a common information element: the configuration template. When reconfiguring an application, the *Configurator* carries out the adaptations decided by the *AdaptationManager* by applying the provided configuration templates, one for each application. These templates contain complete descriptions of the choices made by the *AdaptationManager*, including the composition for each application and their structured subcomponents, blueprints for atomic components, mapping to the nodes in the infrastructure, connectors to use between components, and parameter settings for the components. The *Configurator* uses the information in the configuration templates to create and remove component instances on the local and remote nodes, to connect and disconnect components through local or remote connectors, and to set the parameters of the components.

The *Configurator* maintains information about the current set of running application variants. When requested to perform a re-configuration, the *Configurator* compares the new configuration templates with the information about the currently running applications to derive the (minimal) sequence of steps required to achieve the new configuration.

Before starting a reconfiguration, the current application configuration must be driven to a safe state, e.g. to avoid invalid interactions between application components while component instances

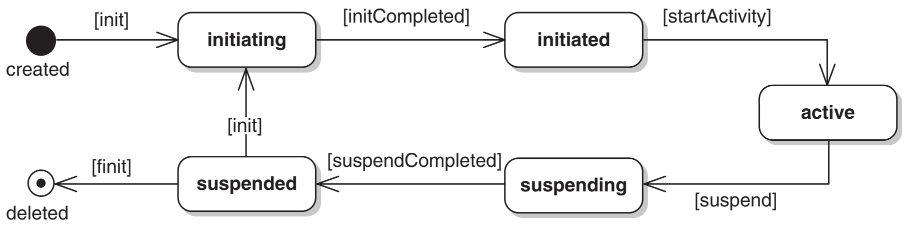


Figure 7. Application component reconfiguration states.

and connectors are being replaced. This requires the collaboration of the components constituting the application. Also state transfer from the old to the new configuration is an important issue. In general, the automation of the required steps, i.e. the detection of dependencies between components and state transfer, are challenging tasks. As these research challenges are considered to be important aspects but out of the scope of the MADAM project, we adopted only a simple solution: The middleware expects configurable application components to implement an interface for changing their configuration state. This interface defines methods for setting the current configuration state of the components, for retrieving the current configuration state, and for transferring serializable internal component state from one component to another when the implementation of a specific component role is replaced by another implementation or moved to another node during reconfiguration. During the reconfiguration, components provide feedback to the *Configurator* when they are finished initiating or suspending. Figure 7 shows a state machine diagram for the configuration states of configurable components. The triggers for the state transfers are methods of the configuration interface of the application components and the feedback interface of the *Configurator*.

3.6. Implementation and prototype

The MADAM project team identified a set of guidelines for the selection of the technical implementation platform. The guidelines have taken into account many factors, such as cross-platform portability, availability of IDE tools, availability of documentation and technical support for the candidate development tools and environments, considerations on performance and pricing, and leveraging existing component-based frameworks and patterns widely adopted by a well-established developers community. As cross-platform portability is one of the main requirements for the MADAM middleware, the reference technologies selected for the development are Java-based, as the Java language facilitates the task of creating platform-independent solutions. However, it was not possible to make all the components completely independent from the platform, as the middleware needs to access low-level physical resources that requires some native code. We will return to portability issues in more detail later in this section.

The resulting MADAM prototype can be deployed and executed on:

- Computing devices that offer a Java run-time environment supporting the J2ME CDC Personal Profile specification. Examples of such kind of environments, where the middleware has been tested, are PDAs running the MS Windows Mobile for Pocket PC operating system, enabled to run Java applications by means of the NsiCom CrE-ME 4.0 Java environment for PDA.

- Computing devices that offer a Java run-time environment supporting the Sun JRE J2SE 1.3.1 specification. Examples of such kind of environments, where the middleware has been tested, are computers running the MS Windows XP operating system, enabled to run Java applications by means of the official SUN Microsystems Java Running Environment 1.3.1 for Windows XP.

After an evaluation phase, the team selected as primary target devices some PDAs of the HP iPAQ family, like the iPAQ 6340 and 6915 running Microsoft Windows Mobile for Pocket PCs. A second target category of devices is formed by Microsoft Windows XP-based laptops and tablet PCs. The selected Java environments are CrE-ME 4.0 from NsiCom Ltd for PDA, which is based on the JDK 1.3.1 specification, and the official SUN Microsystems Java 1.3.1 implementation for Windows XP in order to maintain code compatibility with the mobile Java platform. The development platform is MS Embedded Visual C++ 4.0 plus the MS Pocket PC SDK. The communication between embedded and Java components is based on JNI technology.

Figure 8 shows snapshots of two instances of the MADAM middleware running on two HP iPAQ 6340 PDA and sharing context data. In the following, some development related topics will be described in more detail.

3.6.1. Coding issues

The code for the MADAM middleware was developed by implementing for each component the interfaces as defined in the architectural design. The middleware developers applied a set of well-known and widely adopted design patterns that allow facing implementation issues by means of tested, proven development paradigms. For example, the Singleton pattern was employed for avoiding unwanted and unneeded proliferation of objects instances; the Observer pattern was used to implement subscription for listening to context change events and resource change events; the Adapter pattern allowed to wrap classes and methods from external or native APIs and to offer them in a format compliant with the component interface definition; the Proxy pattern was applied for providing a reference to remote objects.

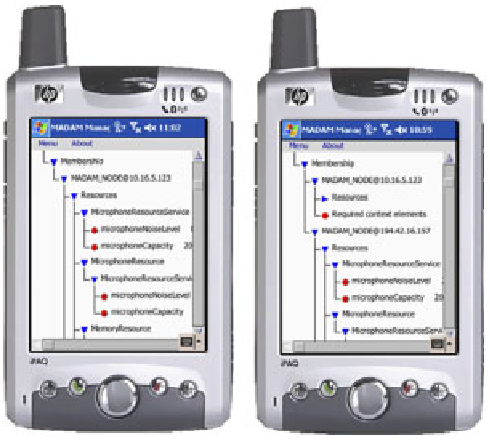


Figure 8. Examples of two MADAM instances running in the same adaptation domain.

In order to optimize the resource consumption and to provide better performance, some of the middleware components create and use Java threads, e.g. for monitoring low-level resources, handling the context distribution, performing adaptation steps, and handling GUI events. As the thread usage may lead to a high consumption of system resources, which is a particularly critical issue on mobile devices, the number of threads has been kept limited and under the control of the MADAM middleware. Thus, they cannot dynamically proliferate in an unmanaged fashion.

The Context Manager runs a custom protocol for distributed context management. The protocol uses multicast messages (based on UDP over IP) for heartbeat signalling. It enables different instances of the middleware running in the same adaptation domain to communicate with each other and to share context data. Driven by the distributed context information provided by the Context Manager, the Adaptation Manager initiates distributed adaptations leveraging the Java RMI technology, which allows to invoke methods on remote MADAM instances. Similarly, the Resource Manager uses RMI to deal with resources of devices distributed over the network.

3.6.2. Network handling

The MADAM middleware prototype was designed to run in mobile environments, where the execution context changes dynamically. In such a scenario, it is fundamental to provide network access mechanisms that enable to automatically detect, select, and use the best network connection among the available ones (if any). Therefore, the middleware relies on a pluggable mobile-IP API, whose tasks are handling transparently the network handover, providing information on the current state of the network connection and on the set of all available network types, and allowing connection to (disconnection from) a given network. The integration of the mobile-IP API into the MADAM middleware takes place in the Resource Manager that provides generic methods for accessing network-related information. A NetworkAdapter class inside the Resource Manager acts as a wrapper for the pluggable mobile-IP API functionalities and provides them to the Resource Manager in a format compliant with its interfaces. The NetworkAdapter class is the only entry point in the whole middleware where external network APIs are accessed.

3.6.3. Cross-platform portability

The selection of a Java-based execution environment for the middleware allows a high degree of cross-platform portability. Nevertheless, as already mentioned, the implementation of some native code was unavoidable to collect information on low-level sensors and resources. The Resource Manager invokes such native code by leveraging the Java Native Interface technology. As a consequence, porting the middleware to platforms other than Windows XP or Windows Mobile requires the development of some custom native code for monitoring those low-level elements that are not directly accessible from Java. The points to modify in the code are anyhow well localized, enabling the developers to integrate new native code without the need for a complete knowledge of the middleware architecture.

An additional modification required for porting the middleware to other platforms is related to the above mentioned mobile-IP API. The middleware was designed to handle the external network API by means of the adapter class as introduced in the previous section. Hence, the usage of a different network API requires a reimplementation of the adapter class.

3.6.4. Lessons learned during development and testing

During the development and testing phases, the middleware development team faced some difficulties that were mainly related to the complexity of distributed environments and the limited resources of the mobile devices. Often these effects are quite subtle and difficult to detect. Let us give two concrete examples.

In order to enable a distributed context data sharing among multiple MADAM nodes running on different devices, it was necessary to reach a compromise between the frequency of context data updates and their propagation over the network and the obvious impacts on the middleware performance. Updates in the context data, in fact, may trigger adaptations of the applications running on the middleware. The adaptation reasoning is a computing intensive task that, on constrained devices, may take full control of the device for a rather long time. When this happens, sometimes the middleware is neither able to receive heartbeat messages coming from remote Context Managers, nor able to generate new ones. As a consequence, it loses its membership to the current adaptation domain, and the whole process of adaptation fails.

Another area that required some extra fine-tuning efforts was the resource management. The low-level resources are constantly monitored and some of them change their state at a high rate. If each such change in a resource would trigger a context change event, it would soon cause an overloading of the Resource and Context Managers. To avoid this, each resource is checked according to predefined policies, e.g. at predefined intervals. Finding the appropriate values for such intervals without affecting negatively the behaviour and the adaptation capabilities of the applications running on top of the middleware required a lot of testing efforts.

4. DEVELOPMENT METHODOLOGY

The development of context-aware, adaptive applications for dynamic operating environments is a challenging task. An important overall goal of MADAM is to facilitate the development of self-adaptive applications. We provide a model-based development methodology that builds on appropriate software modelling elements and automatic model transformations.

4.1. Adaptation model

The adaptation model specifies the variability of the application including architectural constraints, component properties and context dependencies. We use UML 2.0 as modelling language. The standard UML 2.0 meta-model is enhanced by building a UML profile that provides the required modelling concepts for the adaptation capabilities. In order to explain our modelling approach in detail, we demonstrate a step-by-step example using the SatMotion application, which was already introduced in Section 2. As in the previous sections, we show only a simplified version here.

The variability model starts with the specification of different possible realization options of the application at the top level. In our example the application can be realized through any of the three composition plans shown in Figure 9.

Each of the composition plans describes—among other things—a specific composition of component types that collectively realize the application. Figure 10 shows the composition of component types for the 'Standard' composition plan of Figure 9.

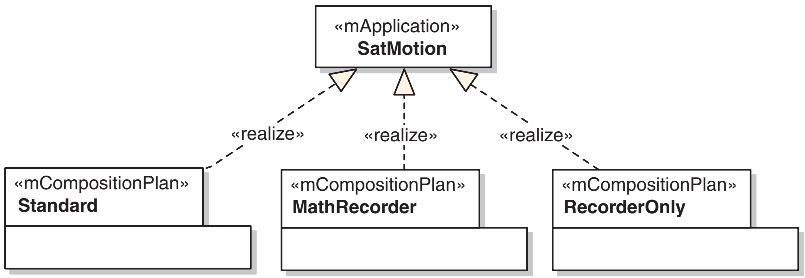


Figure 9. Different realization plans for the application.

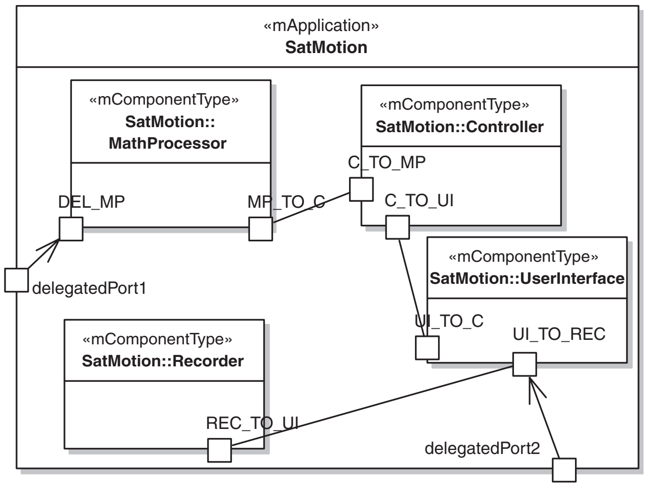


Figure 10. A possible composition of components for the SatMotion application.

The composition consists of four component types (marked with the <<mComponentType>> stereotype): Controller, MathProcessor, Recorder, and UserInterface. Therefore, just like the application each of the component types may have a number of different realization plans. For example, the Controller component type has three different realizations (see Figure 11). A composition plan again consists of a composition of component types, whereas a blueprint plan describes the realization of a single component. Different variants of the application are created by resolving recursively all possible variation points; the recursion ends once a blueprint plan is reached.

Resolving all the variation points with all the possible options can effectively create a huge number of different application variants, all of which have to be evaluated at runtime in order to find the variant that is best for a given context condition. However, very often not all variants are actually feasible. For example, the selection of a particular realization for a component may imply a certain realization for another component. Likewise, a particular realization may be incompatible with some other component realizations. In order to ensure appropriate application variants for all context situations, infeasible variants have to be eliminated. Furthermore, the potential

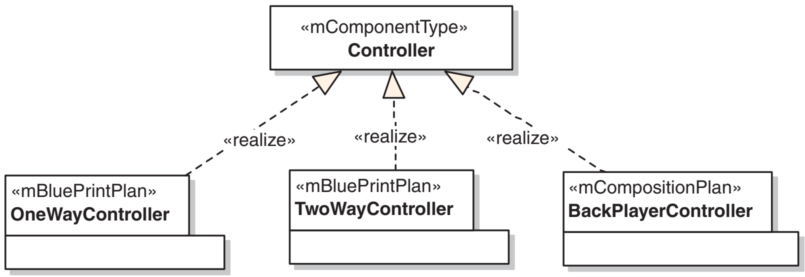


Figure 11. Different possible realization plans for the Controller component type.

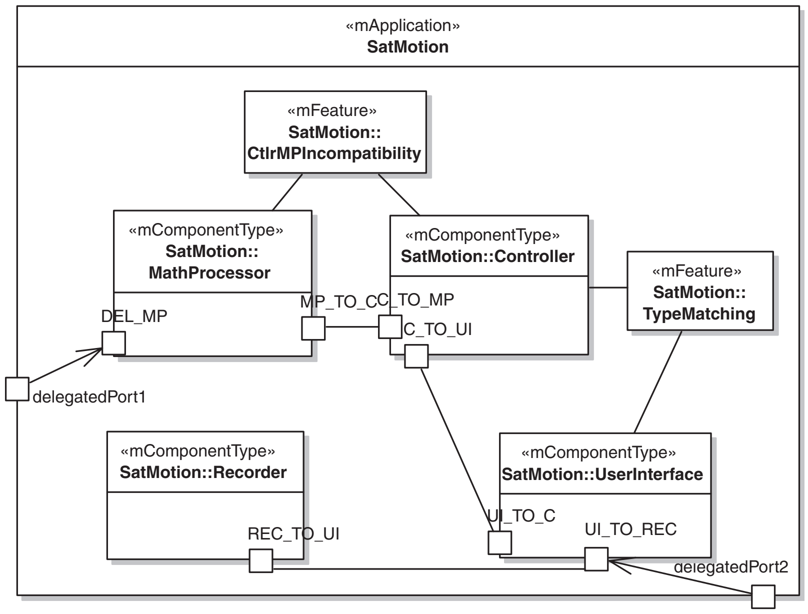


Figure 12. Architectural constraints through feature specifications.

combinatorial explosion of variants can lead to a scalability problem requiring too much computational effort for a resource-scarce mobile device [20]. Therefore, we have added means to filter out the infeasible combinations. This can drastically reduce the number of variants to be considered at runtime. Our filtering approach builds on architectural constraints specified as features in the application model [21]. An example of a composition with architectural constraints is presented in Figure 12. The feature *CtrlMPIncompatibility* prevents the selection of incompatible components for *MathProcessor* and *Controller*; the feature *TypeMatching* enforces that only components with compatible types are selected for *Controller* and *UserInterface*. Feature associations at the composition level just indicate which components in the composition have to be evaluated with regard to a certain feature. The actual constraints indicating if a component feature requires or excludes features of other components are specified at the atomic level, i.e. in the blueprint plans. Thus, at first sight the architectural constraints appear to be static as the feature associations are fixed. However, they can also be evolved dynamically, as new plans—composition plans and blueprint plans—with their own feature associations and specifications can be deployed at run-time.

The adaptation model also specifies the context and resource dependencies of the components, the component types and eventually the application. Both, context entities (and their types) and resources (and their types) are modelled hierarchically, including their offered and required properties. The context entities and resources are associated with the components and component types, and thus the context and resource dependency is established. An example resource model is presented in Figure 13. It shows resource types such as Memory, Network, and Battery and their respective properties. Context entities are modelled in the same way. For a more detailed discussion the reader is referred to [15].

Adaptation in MADAM is built on compositional adaptation that enables the context-aware exchange and reconfiguration of application components at run-time. Thus, application algorithms and strategies can be modified by integrating components that may not have existed at application design time. Note that this approach supports unanticipated adaptation insofar as new component realizations may be added at runtime to the component framework, thus effectively enlarging the number of variants dynamically.

Compositional adaptation is a much more powerful technique than parameter adaptation where parameter values of components are adjusted as a reaction to context changes [13]. Nevertheless, we wanted to support parameter adaptation. Otherwise the developer might end up with a cumbersome

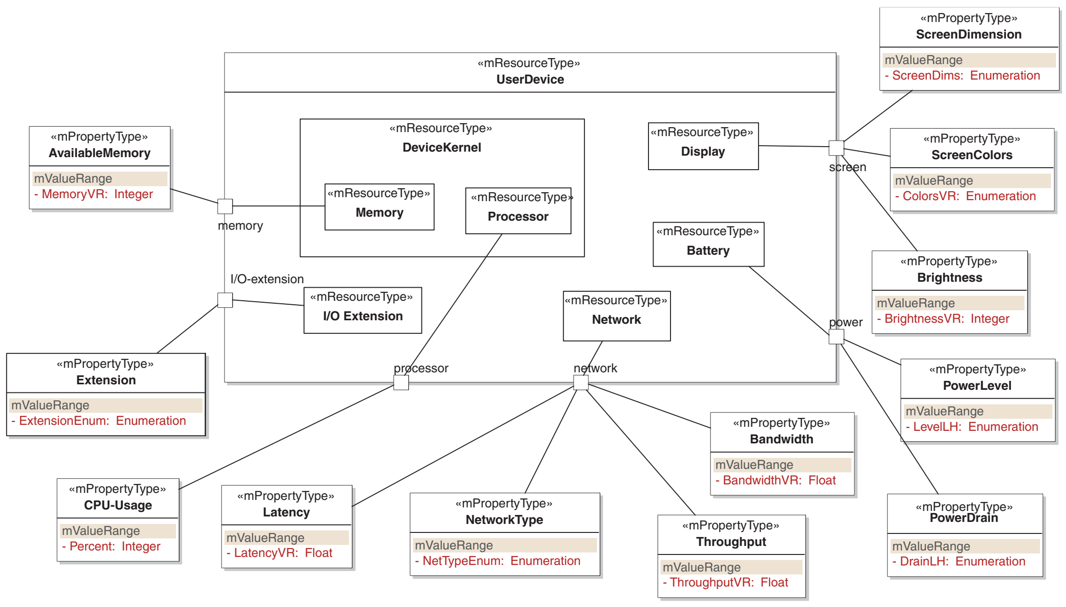

«mResourceType» UserDevice
«mResourceType» DeviceKernel
«mResourceType» Memory
«mResourceType» Processor
«mResourceType» Display
«mResourceType» Battery
«mResourceType» I/O Extension
«mResourceType» Network
memory
I/O-extension
processor
network
screen
power
«mPropertyType» ScreenDimension mValueRange - ScreenDims: Enumeration
«mPropertyType» ScreenColors mValueRange - ColorsVR: Enumeration
«mPropertyType» Brightness mValueRange - BrightnessVR: Integer
«mPropertyType» PowerLevel mValueRange - LevelLH: Enumeration
«mPropertyType» PowerDrain mValueRange - DrainLH: Enumeration
«mPropertyType» AvailableMemory mValueRange - MemoryVR: Integer
«mPropertyType» Extension mValueRange - ExtensionEnum: Enumeration
«mPropertyType» CPU-Usage mValueRange - Percent: Integer
«mPropertyType» Latency mValueRange - LatencyVR: Float
«mPropertyType» NetworkType mValueRange - NetTypeEnum: Enumeration
«mPropertyType» Throughput mValueRange - ThroughputVR: Float
«mPropertyType» Bandwidth mValueRange - BandwidthVR: Float

Figure 13. Resource model.

explicit modelling of different parameter values as different component configurations. Therefore, we have integrated convenient modelling support for component parameters that may assume a range of (discrete) values. Internally, the middleware treats parameter adaptation just like compositional adaptation: The parameter model is transformed to different component compositions whose utility is evaluated by the Adaptation Manager separately.

In summary, the MADAM variability model captures all variation options as well as their utilities and context and resource dependencies in a concise, flexible, and easily readable way. The platform-independent variability model is automatically transformed by a chain of tools into code that the MADAM middleware uses to perform adaptation decisions.

4.2. Model transformation

In accordance with model-based development, the application adaptation model is built as a platform-independent model using a UML modelling tool. It is then transformed by means of a model-to-text transformation into code that performs adaptation-related functions such as evaluating context parameters and resource conditions. The adaptation middleware calls this code at runtime when resolving the variation points and evaluating the utility of possible variants. If there is a context change during application execution, the underlying adaptation middleware computes on-the-fly all possible application variants and evaluates their fitness with respect to the current context situation. The 'best' variant is selected and instantiated [22].

In MADAM we have used the Eclipse Modelling Framework, in which UML modelling tools such as Omondo and Borland Together Architect can be integrated. The UML application model is exported to XMI [23] in accordance with the meta-model defined by the Eclipse UML2, which is a lighter version of the OMG UML 2.0 specification. The UML2 model exported as XMI is taken as input to generate programming language code using MOFScript [24], which is an implementation of OMG's model-to-text standard [25]. MOFScript comes as an Eclipse plug-in. The generated code is then published to the middleware. Figure 14 illustrates the tool chain: The platform-independent adaptation model (PIM) written in UML is transformed via an intermediate transformation into XMI and finally into a platform-specific model (PSM) which in this case targets Java.

Development of the abstract, platform-independent model and performing automatic code generation provides high flexibility for the development of adaptive applications. If another target-platform should be addressed, the abstract high-level model can be reused and only the transformation has to be adjusted to meet the needs of the new platform. Furthermore, if changes in the overall structure of the application are necessary, the modifications can be done at the higher abstraction level of the model and the corresponding code is generated automatically. Abstract system specifications are also very useful to keep track of the set of possible application variants and to ensure completeness.

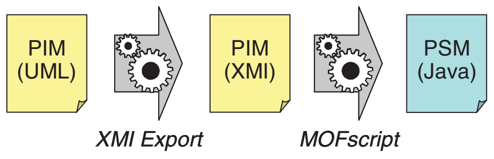


Figure 14. Model transformation tool chain.

5. LESSONS LEARNED FROM CASE STUDIES

Two realistic case studies were developed to evaluate the MADAM approach. Particularly, we wanted to find out about the development overhead for building an adaptive application and the behaviour of the resulting adaptive system as perceived by the end user. The assessment of the development process was based on the comparison of the MADAM platform with commonly used methods, languages, and tools. The assessment of the resulting system was based on aspects such as usability, performance, processing overhead, and memory footprint.

5.1. Case studies

In the first scenario, which was derived from a real application of Condat AG written for Daimler Chrysler AG and Deutsche Bahn AG, service technicians are supported during their scheduled tasks regarding inspection, measurements and maintenance of technical equipment such as air conditioning plants or fire alarm installations. The technician uses a PDA for his work. In the original version of the application he had to handle different operating modes manually. With MADAM, when the technician's environment changes during his roundtrip, the application on the PDA tries to self-adapt automatically in order to retain its usefulness. For example, in the original version of the application when a communication link deteriorated the technician had to wait until the connection was up again. Using MADAM, the application automatically tries to transmit the data using alternative connections or buffer them temporarily on the local device.

The second pilot application is called SatMotion and was written by Integrasys SA. It is a distributed application that supports the remote line-up of satellite antennas by taking care of power level and receive signal. The adjustment is performed using a PDA that is used to control the measurement equipment and a server over a wireless connection. The application receives the signal traces, visualizes and evaluates them using a spectrum analyzer. The results allow the installer to find the optimal alignment of the satellite antenna. In the original application, depending on the installer's tasks and context conditions, the installer could switch manually between different operating modes. In the new version based on MADAM, the application automatically selects an appropriate operating mode: For example, the *TwoWay* mode enables to command the remote instrument and receive signal traces, the *OneWay* mode only receives data for visualization, and the *Offline* mode is able to perform measurements and generate reports on recorded information. If, for example, the network context elements bandwidth and delay deteriorate beyond a certain threshold, the system will switch from *TwoWay* to *OneWay* mode in order to maintain the usability of the application.

In addition, several smaller test applications were written that can be parameterized to measure performance, memory, and scalability.

5.2. MDD approach

The modelling language and tools of the MADAM framework enable the application developer to specify adaptation capabilities at design time and to generate executable code for the runtime environment. The developer models the structure of the pilots in terms of nodes, components, resources, sensors, and adaptation capabilities and defines a utility function that captures the goals of the adaptation and ensures that the active application configuration always has the highest utility regarding the resource constraints imposed by the environment. The completed adaptation model is automatically checked for consistency and transformed into a platform-specific format that has to be loaded into the MADAM run-time environment before application execution. Note that in MADAM adaptation decisions do not consider the overhead caused by the (possibly distributed) adaptation. This interesting question is left for future work.

In parallel to the adaptation modelling, the component-based application has to be developed, in our case in Java. Obviously, adaptation model and application are highly interrelated, because they refer to the same set of components, context sensors, and resources. Thus, the developer must make sure that all interrelations are defined consistently. During application run-time the MADAM adaptation middleware monitors the context and adapts the application in response to context changes according to the structure and adaptation rules reflected by the platform-specific adaptation model.

The development of the adaptive pilot applications required new considerations for the system design. How should the application be divided into components? How can the developer map the intended adaptation strategy onto a utility function? Adding adaptivity to an existing application usually requires a re-engineering of the application architecture. Even if the application is already structured in components, a redesign of the component composition and interfaces is often necessary in order to achieve all needed variants. As the MADAM approach always implies some overhead in terms of development effort and run-time, it is targeted primarily at distributed applications that have a larger number of context dependencies and adaptation options. One can argue about whether MADAM is beneficial for simple, small applications that depend on only one or two context elements.

During the development of the case studies it turned out that some developers found it rather difficult to assign appropriate properties to components and to determine the utility function for an arbitrary application domain. The developer must spend some time to find the right strategy, the property assignments and the utility function for the required adaptations. The best way for the validation and refinement of the selected strategy is often by implementing a pilot application. The adaptivity of an application, i.e. the suitability of the adaptation model and the defined utility function, can be tested and evaluated using a scenario engine, which was built for MADAM. It is script-driven and simulates context change events. (See the following section.)

In addition, the developer needs some experience to find an efficient component architecture, especially for applications with a lot of variants (e.g. $>10\,000$). A well-chosen component structure and the usage of architectural constraints in the modelling can significantly reduce the amount of variants and improve the performance.

How do you evaluate a development methodology as such? Obviously it takes a number of empirical studies in order to 'proof' that a certain methodology prevails over other ones. A research project such as MADAM cannot answer this question sufficiently. After all, it takes years of practical experience with a software development methodology in order to build up confidence in its benefits and to realize its drawbacks. Nevertheless, our experiences with model-driven development approaches are definitely positive—not only for MADAM but also in other research projects.

5.3. Prototype performance

In order to assess the runtime overhead of the MADAM middleware we have performed a set of measurements using the two pilot applications as well as special measurement applications. In this section, we first describe briefly the applied metrics and how the measurements were performed, and then summarize and discuss the main findings.

Relevant metrics were:

- Memory footprint, i.e. the memory required to store the middleware code and the context, architecture and property prediction models used by the middleware at runtime.
- Context monitoring overhead, i.e. the additional processing needed to monitor the context and detect context changes.
- Context latency, i.e. the time from when a context change occurs until it is signalled to the Adaptation Manager.
- Planning time, i.e. the time needed by the middleware from when a context change is signalled to the Adaptation Manager until an action has been decided. During this time the middleware competes with the applications for CPU time.
- Reconfiguration time, i.e. the time from when the planning has been completed until the new configuration is operational. During this time the applications are blocked. Clearly, this time varies with the complexity of the reconfiguration to be done.

In order to ease the isolation of the effect of the middleware from the effect of the normal application execution during measurements, in addition to the two real adaptive applications we developed several special measurement applications that were basically skeletal applications with specific adaptation models such that they could be adapted by the middleware, but demanded negligible resources to execute. This included

- an ‘empty application’, with no variation and no context dependencies;
- a ‘tiny application’, with one variation point with two alternatives and one context dependency;
- a ‘skeleton application’, which is built as a composition of empty components of the same type, which can be scaled in terms of the depth of the composition hierarchy, the number of roles of each composition, and the number of alternative implementations available for each role.

Not all metrics could be measured directly but had to be calculated from more primitive measurable metrics. In addition, there are sources of error that could disturb the accuracy of the measurements. However, by instrumenting both the pilot applications and the measurement applications appropriately and running a number of test cases repeatedly, we were able to obtain measures for the metrics we were interested in, which we judge to be sufficiently accurate to assess the feasibility of the MADAM technology. It should be noted here that (a) all reported measurement results are average values obtained from several repeated experiments, and (b) that the three measurement applications and the two pilot applications are all different from each other in terms of size and structure of adaptation models, complexity of utility functions, and degree of distribution of the application. This is the reason why the measurement results for the different application types differ substantially. More details about how the measurements were performed can be found in [5].

To aid the measurements we used a scenario engine as illustrated in Figure 15. The scenario engine is driven by scripts and simulates context changes and user input, which is fed to the system under test. Thus, we can easily get insights into the adaptive behaviour of the application for a variety of context conditions as well as in the effectiveness and correctness of the chosen utility functions. The executed scenario and the output from the system were recorded in a log file. The

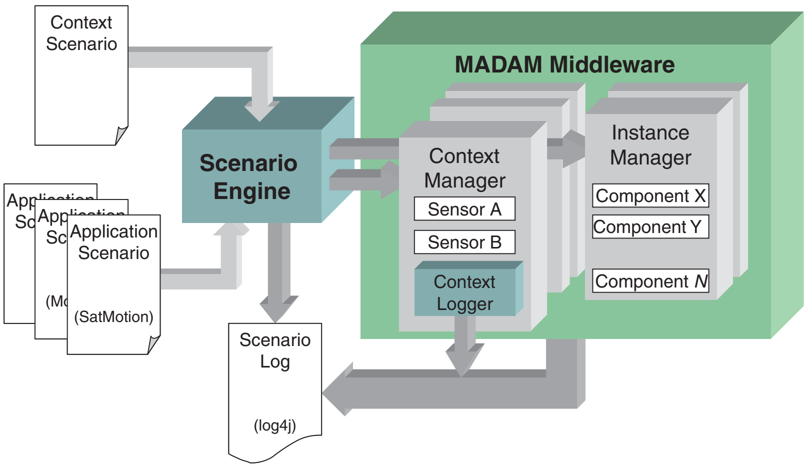


Figure 15. Testing environment.

hardware environment for the tests consisted of a HP iPAQ PDA and some desktop PCs connected by a wireless LAN. Details are given in Table I.

The main results from the performance measurements are shown in Table II. We performed the measurements on two different configurations, one with the PDA acting as master and two PCs as slaves (column 'PDA'), and one with a desktop PC acting as master and two other desktop PCs as slaves (column 'Desktop PC'). Obviously, context monitoring overhead and context latency depend on the number and nature of the context properties to be monitored. We measured them for the set of context properties that the pilot applications were sensitive to.

As already mentioned above, the pilot applications were derived from real products. One has 12 variants and the other one has 180 variants. We believe that they are fairly representative for the kind of applications we have to deal with on handheld devices, both in terms of size, context dependencies, and adaptation complexity. Needless to say, there is only one active user per mobile computing device.

The memory footprint clearly depends on the number of active applications and their adaptation complexity. However, the effect was small. The measured memory footprint is in the order of 10% of the available memory on the PDA that was used for the tests, which is not unacceptable.

With one empty application running on the handheld device, the extra continuous load on the CPU caused by the MADAM middleware is 13%. The adaptation time, i.e. the sum of the planning time and the reconfiguration time, which is the period when the operation of the application may be more severely disturbed, is in the order of a few seconds. We believe that a few seconds delay now and then is a price that most users would be willing to pay—provided that the effect of the adaptation is observable and positive. Note that a detailed investigation of such human-computer interaction issues was beyond the scope of MADAM, but would definitely make an interesting topic for future work.

On the PDA, one pilot application together with the middleware almost filled up the memory, so scenarios with multiple applications running concurrently are not likely on this class of devices.

Table I. Devices used for the measurements.

Device	Processor	Memory	OS	Java VM
Desktop PC	Pentium 4 2.4 GHz	1 GB	MS Windows XP SP2	JDK 1.3.1_15
PDA HP iPAQ 6345	Texas instruments 68 MHz	64 MB	MS Windows Mobile 2003	CrèMe 4.00

Table II. Selected measurement results.

Metric	PDA	Desktop PC
Memory (base)	6–8 MB	6–8 MB
CPU (continuous)	13%	7%
Context latency (local)	190 ms	23 ms
Context latency (remote)	287 ms	70 ms
Planning (skeleton test application, 100 variants)	787 ms	87 ms
Planning (skeleton test application, 1000 variants)	1125 ms	112 ms
Planning (pilot 1, 12 variants)	742 ms	87 ms
Planning (pilot 2, 180 variants)	1300 ms	430 ms
Reconfiguration (tiny test application)	715 ms	94 ms
Reconfiguration (pilot 2)	4400 ms	1468 ms

However, more recent devices offer significantly more resources, typically 10 times more for both processing and memory capacity. On such devices we can envisage scenarios with several applications running concurrently and more complex distributed computing environments. A typical scenario might be three applications similar to our pilot applications, with three auxiliary computers available for running components, the possibility to choose between WLAN, UMTS, and GPRS for the communication and three settings for the CPU frequency of the handheld. Extrapolating our measurements to this scenario, we also get adaptation times in the order of a few seconds.

In our measurements the planning time and the reconfiguration time contributed roughly on equal terms to the total adaptation time. With a further increase in the number of applications and variation points we quickly move into a steep area of exponential growth of the planning time with the number of variation points, which causes the planning time to dominate and is likely to lead to unacceptable adaptation delays.

5.4. Required improvements

The validation of the MADAM framework by the pilot applications has shown that the general approach is viable and successful. However, some improvements and enhancements are necessary, which will allow the MADAM middleware and methodology to serve as a foundation for future developments.

The modelling tools and run-time environment were evaluated in terms of usability, learning effort, flexibility as well as for the scope and completeness from the developer perspective. One main result of the evaluation was that the modelling approach for self-adaptive mobile systems leads to a systematic and principled way, avoiding partial, *ad hoc* solutions. However, the modelling concepts represent sophisticated emerging technology. When designing adaptation models for the pilot applications, the developers asked for more insights into the modelling language and the interrelation between the model elements and the generated code. In a way this contradicts the purity of the model-driven development paradigm but reflects the necessities of real life. In addition, developers wanted more support, i.e. methods and tools, for finding and defining appropriate property predictors and for assigning utility values to component configurations.

Regarding flexibility and completeness, the offered MADAM functionality has proven to be sufficient for implementing distributed and multiple adaptive applications. One potential improvement has been identified: as a fixed load order of distributed MADAM applications is required at present, a more flexible launch should be supported.

Currently, the MADAM middleware prototype allocates about 7 MByte memory, which is acceptable for a PC, but a bit high for a PDA. The performance is good on the PC, but not always on a PDA that runs already loaded large applications. In future work, we intend to improve performance and memory footprint, because this would increase the user acceptance and offers the opportunity to apply MADAM on even smaller devices (e.g. smart phones, multi-functional MP3 players, smart home devices) and to support applications comprising more concurrent processes or variants.

Although the tests indicate that interesting and realistic scenarios can be tackled also on handheld devices without disturbing the normal execution of applications unacceptably, performance improvements both regarding memory footprint and adaptation times are desirable. In particular there is a need to reduce the planning time. An obvious improvement concerns the preciseness of the variability model. In an application normally not all alternatives of all variation points can be combined freely. Typically, there are dependencies between variation points, such that choosing one alternative for one variation point constrains the choice at other variation points. This can be expressed as architectural constraints in the MADAM modelling notation but currently the middleware does not exploit this. Therefore ‘illegal’ variants have to be suppressed by ensuring that their utility is zero. This means that during planning the middleware considers more variants than necessary. Often architectural constraints reduce the number of variants to consider substantially. For example, in the pilot with 180 variants the total number of ‘legal’ variants is only 28. Therefore, we believe that leveraging architectural constraints in the middleware may reduce the planning time significantly.

Another possibility is to delegate the planning to a powerful server. Our tests indicate that even on an average laptop, planning is 10 times faster than on the handheld. Delegation of planning will work only when the device is connected, so there must be an onboard backup that takes over planning while the device is disconnected. However, as when a device is disconnected there are normally much fewer application variants that are possible than when it is connected, this approach may also have a significant effect on reducing planning time.

Finally there is the possibility to introduce heuristics in the planning algorithm that is able to make a decision without having to consider all variants. In particular if we relax the requirement to always find the optimal variant, there are good opportunities for gaining performance.

In conclusion we think that the development and testing of the pilots have demonstrated the feasibility of the MADAM approach. Adaptations were always meaningful and useful, and the development and runtime overhead were acceptable. In some cases, it would be helpful if significant reconfigurations were indicated suitably to the user, e.g. if an important function is no longer available. Another helpful option would be, if the professional user had means to select variants himself.

6. RELATED WORK

Development methodologies, platforms and middleware supporting dynamic adaptation of context-aware applications on mobile computing devices have been studied extensively during the last decade. In this section, we compare the work of MADAM to a few other representative and related efforts. The diversity of research in this area is vast as illustrated in [2,26]. For this work we are concerned about related efforts on development methodologies and dynamic middleware platforms for adaptive applications. Clearly, MADAM has not been the first project to tackle dynamically adaptive software systems. It is based on incorporates and extends earlier research work. However, MADAM stands out as it provides a truly comprehensive solution that addresses adaptation from both the theoretical and the practical perspective and solves many challenges in one single integrated framework.

Early adaptive systems addressed QoS concerns and QoS management involving maintenance and renegotiation of QoS—but these were not really context-aware and not targeting mobile computing [27]. The emergence of adaptive middleware generalized earlier approaches to adaptation by providing open implementations and reflection as a general means to adaptation. Notable examples include the ADAPT project [28] introducing open bindings as a means for adapting video stream bindings, OpenORB [29] introducing meta-level objects and causal connections between the meta and base level in reflective middleware, and dynamicTAO [30] supporting dynamic configuration of the ORB itself. However, these works are still very much QoS-oriented, focusing on adaptation of middleware level services. In later years, there has been an increasing interest in context-awareness and mobile computing focusing on application-level adaptation. MADAM falls under this category of projects. A comprehensive survey of projects on adaptive software systems is provided in [13]. Below we compare the MADAM framework to some related projects in this domain.

The Bay Area Research Wireless Access Network (BARWAN) project ([31,32]) at the University of California Berkley, is an early project that combines overlay networks, proxy servers, and reliable data transport for wireless networks. The overlay network provides seamless roaming between mobile communication systems and hand-over between WLAN cells. The proxy servers dynamically adapt retrieved content based on context. In MADAM, an open API was defined for interacting with mobile IP clients that handle the roaming and handover, and through this API network properties are monitored and collected for use in the adaptation reasoning. MADAM does not predefine the use of proxy servers, but such could be included as part of the architecture of the adaptive applications. In the utility-based approach of MADAM, the decision on how to configure a proxy server and whether to use one at all would be made based on what gives the best overall behaviour of the system.

CARISMA [33] is a mobile computing peer-to-peer middleware that exploits the principle of reflection to support the construction of context-aware, adaptive applications. In contrast to MADAM, 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. Although the adopted rule-based policies are natural and simple to use, there are some drawbacks compared with MADAM'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 MADAM, the lower level reconfiguration actions are automatically determined by the middleware. Adaption in MADAM is QoS (or property) driven while the CARISMA rule-based policies do not consider prediction of non-functional properties. Lastly, CARISMA handles only a fixed number of adaptation policies whereas the utility function approach of MADAM 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.

Chisel [34] is an open framework for dynamic adaptation of services using reflection in a policy-driven, context-aware manner. It is based on decomposing the extra-functional aspects of a service into alternative behaviours. Then, a policy driven algorithm is used to dynamically select the most suitable alternative, based on the context changes. Similar to MADAM, context changes trigger a process that intelligently determines if an adaptation is required. Unlike MADAM which uses utility functions, Chisel leverages human-readable declarative adaptation policies to instruct the adaptive behaviour of the applications. These policies are defined with a custom-made language and in contrast to MADAM are essentially rule based.

MobiPADS [35] enables active service deployment and reconfiguration in response to a dynamic environment. This is achieved with the use of so-called mobilets, which are active pieces of mobile code existing in pairs: a primary mobilet executing on the client (mobile) node, and a secondary mobilet executing on the server side. The mobile client offloads as much of the computation and storage burden as possible to the server side. Once a context change is detected, the MobiPADS middleware either reconfigures the current mobilet service chain, by notifying the mobilets about the context changes so they can internally adapt themselves, or by instructing the applications to adapt. Although the intentions of MobiPADS are similar to the ones of MADAM, obviously they have chosen a different technical approach.

Odyssey [36] is a QoS-middleware that dynamically chooses a version of the content before transporting it across the wireless network. Odyssey is running on the client side and the content is stored on the server side. Applications on the mobile computer run independently from each other. The adaptation decisions depend primarily on the quality of the transmission link. Applications provide a window of tolerance, i.e. a lower and upper bound on the required resource quantities. If resource capacity changes beyond these thresholds, the application will be notified. Application-specific Wardens capture the adaptation possibilities of adaptable applications and offer a choice of adaptation levels. Rather close to the principle of the utility function of MADAM, Odyssey introduces the idea of fidelity to label different versions of the same content stored on a server. Fidelity is the degree to which the content presented matches the original reference copy. A fidelity level is selected by mapping monitored QoS values of the network to fidelity levels.

Quality Objects (QuO) [37,38] is a well-known middleware project. Although QuO does not address the fundamental problems associated with portable devices and wireless networks, it has had a major impact on adaptive and QoS-aware middleware development projects, including MADAM. QuO is an architecture that focuses on providing design-time and run-time support for adaptive client–server applications. The QuO approach enables applications to adapt their functionality dynamically to changing communication and computing resources. It requires run-time monitoring of the resources and a detailed description of all possible adaptations in each application component. QuO contracts define the possible application adaptations under changing resource characteristics. QuO contracts define a QoS space that is divided into regions that are defined by predicates on system properties. When system properties change, QoS can move from one region to another, which may trigger adaptive actions to keep a certain quality level. A disadvantage of the QuO architecture is that it only focuses on adapting the application's internal functionality based on the availability of system resources. QuO does not consider adaptations in the deployment configuration of a distributed application. QuO (in addition to MADAM) is also one of a few projects that have attempted to address adaptation as a reusable component or service. Specialized QoS management functions are encapsulated by so-called Qoskets. Qosket components provide a general solution to manage end-to-end QoS requirements without requiring the middleware to provide such services. However, QuO does not provide a framework that is able to reason about the needs for QoS management functions at run-time, and to map these needs to concrete Qosket components and patterns for how to use them.

The QuA project [39] has developed a QoS-aware adaptation middleware that is based on many of the same principles as the MADAM adaptation management, i.e. using run-time models, utility functions, QoS prediction, and service planning. In contrast to MADAM, QuA provides a generic middleware level service that can be used to retrieve reflective models, referred to as service mirrors of applications and services in any phase of their life cycle, including run-time. Service mirrors are maintained by the middleware to reflect the life-cycle state of the entities they model. QuA does not provide a comprehensive context management middleware but may exploit a context middleware similar to the one provided by MADAM. The QuA middleware does not focus on mobile applications, but rather on supporting adaptive multimedia streaming applications. A related effort, QuAMobile [40] has tuned the QuA core middleware towards context-aware mobile computing systems. QuAMobile introduces a QoS-architecture and a distributed meta-level representation of context and services for mobile computing. The QoS architecture of QuAMobile has been validated as a proof-of-concept implementation, and thus no complete prototype has till date been realized.

The self-adaptation techniques proposed by the Rainbow approach [8] are similar to MADAM in that they attempt to separate adaptation from application logic concerns. Rainbow extends architecture-based adaptation by adopting architectural styles. Architectural styles are used to tailor the adaptation infrastructure by encoding the developer's system-specific knowledge, including adaptation strategies and system concerns. However, Rainbow makes no attempt to tackle mobile environment requirements. Furthermore, its developers appear to have based the adaptation strategies on situation-action rules, which specify exactly what to do in certain situations. In contrast, MADAM uses extended goal policies expressed as utility functions, which is a higher level of adaptation strategy specification that establishes behavioural objectives, and which leaves the system to reason on the actions required to implement those policies.

ReMMoC [41] is a dynamic middleware that supports interoperability between mobile clients and ubiquitous services. During run-time, the ReMMoC service discovery component reconfigures itself and the remote service binding to match the protocols of the discovered ubiquitous services. ReMMoC provides an abstract programming model that alleviates some of the problems associated with designing applications for dynamic middleware platforms. The programming model offers generic APIs that are mapped to the technology-specific APIs. Like MADAM, REMMoC uses architecture specifications for both the initial configuration and reconfigurations. ReMMoC does not support anything like service planning or search for service implementation alternatives, but applies rule-based policies limited to a fixed set of static component compositions.

UbiQoS [42] is a context-aware middleware that adapts media content according to QoS requirements and portable device characteristics. The middleware assumes a proxy and a QoS-adapter in the wireless network. QoS requirements of different users and context information (including network QoS) about the mobile terminals are stored in a centralized server. This information is used by the adapter to adapt the content to fit the portable device and the network QoS, and to satisfy the user QoS requirements. UbiQoS does not implement anything like generalized run-time service planning as in MADAM but focuses on QoS maintenance of media content. Furthermore, UbiQoS implements context sensing, but does not provide context information management interfaces to applications or other middleware functions as in MADAM.

Regarding the development methodology for context-aware, adaptive applications, the research reported in [43,44] has some similarities to the MADAM approach. Their approach also emphasizes the importance of models. A rich set of conceptual models for modelling context dependencies and adaptation is proposed to support the software engineering process. Adaptation is controlled by defining context-dependent preferences for the application behaviour. A software infrastructure is described that supports the context-awareness and adaptability of applications. The overall objectives are almost identical to MADAM. However, MADAM has chosen a development methodology that builds heavily on the notion of dynamically reconfigurable component frameworks, adaptation decisions based on dynamically computed utility functions, and an MDA-compliant development process based on UML models and popular MDA tools. Furthermore, MADAM supports unanticipated adaptations by enabling the introduction of new components at run-time.

7. CONCLUSIONS AND FUTURE WORK

Run-time adaptation will be a major requirement for future software systems in ubiquitous computing environments. Technologies such as mobile computing, *ad hoc* networking, pervasive computing infrastructures, and dynamic service discovery clearly contribute to this trend. The MADAM project has shown the viability of a general and comprehensive approach to the development of self-adaptive applications for mobile computing scenarios. MADAM provides a computing infrastructure as well as a development methodology for novel applications that are able to react to context changes and adapt their behaviour in order to maintain their usefulness under different execution conditions. Self-adaptation will lead not only to more useful and more versatile but also to more robust and more dependable applications.

Looking back to the initial reference requirements as listed in Section 2.1.2, MADAM has achieved significant results. The applicability of the proposed solution to the complete set of the reference requirements was studied and found to be satisfying to all of them [5]. Because of resource and time constraints, not all the requirements were experimentally tested, however. For instance, the *User and application session redeployment* and the *Security* adaptations were not included in the pilots. Nevertheless, most of the requirements were experimentally evaluated in the case studies: *User Interface presentation* by adjusting the volume of a janitor's PDA when a plane flies by, *Network availability* by automatically switching from WiFi to GPRS when the former becomes unavailable, *Data replication and synchronization* by automatically enabling an offline mode when the network is inaccessible, etc. [4]. Adding to the soundness of the theoretical results of the project, the experience with its practical conformance to the reference requirements has increased our confidence in MADAM as a comprehensive solution for developing context-aware, self-adaptive applications.

On the one hand MADAM has tackled successfully a rather broad range of research issues. On the other hand it has opened up our eyes for a number of new challenges that wait to be solved. For example, usability and human-computer interaction are important issues where we need more insights. Will the user accept systems that adapt automatically? How often may an application adapt without becoming an annoyance to the user? Does the user trust a system that changes automatically? This leads to another open problem area: How do you validate and test a self-adaptive software system that is able to handle unanticipated adaptation at run-time? How about security and trust in distributed context information? Many of these questions can only be answered through experiments and practical experience with self-adaptive systems.

Furthermore, as the inception of MADAM several years ago new technological achievements have introduced additional requirements and opportunities. For example, *ad hoc* networking facilities and ubiquitous service architectures are made available that represent an enrichment of an application's execution context. Thus, an adaptive application may want to replace a local component by a remote service if it promises a better service. We need new context models and context query languages to model these environments and fully exploit such scenarios. The usage of remote services must be controlled by some implicit or explicit service level agreement. Adaptation decisions may depend on the quality of a service as well as on its price. It has to be explored what kind of decision support techniques are appropriate for controlling such adaptations. Even more open questions arise when we also allow the adaptation middleware to adapt dynamically.

We have recently started a new research project called MUSIC which addresses many of these open questions by building on the results of MADAM and extending them towards ubiquitous computing scenarios. Thus, the ideas underlying the MADAM project represent a long-term and continuing research thread in the realm of self-adaptive software.

ACKNOWLEDGEMENTS

The contributions of all members of the MADAM team are gratefully acknowledged. We thank the anonymous reviewers for their constructive comments. MADAM was supported financially by the EC under contract FP6 IST 4159.

REFERENCES

1. MADAM Project Home Page. http://www.intermedia.uio.no/confluence/display/madam [4 August 2008].
2. Aksit M, Elrad T (eds.). Special Issue: Experiences with auto-adaptive and reconfigurable systems. *Software—Practice and Experience* 2006; **36**(11–12):1227–1229.
3. Chimaris A, Papadopoulos G, Paspallis N, Abraham Z (eds.). *MADAM Deliverable D1.2*, *Adaptation Scenarios and Reference Requirements for Service Adaptation*. http://www.intermedia.uio.no/confluence/display/madam/Deliverables [4 August 2008].
4. Ruiz P, Sanchez J, Fricke R (eds.). *MADAM Deliverable D5.3*, *Pilot Services Adaptation Requirements and Design*. http://www.intermedia.uio.no/confluence/display/madam/Deliverables [4 August 2008].
5. Fricke R, Klimasek J, Sarioglu A (eds.). *MADAM Deliverable D6.2*, *Pilot Services Trial Evaluation*. http://www.intermedia.uio.no/confluence/display/madam/Deliverables [4 August 2008].
6. Floch J, Stav E, Hallsteinsen S. Interfering effects of adaptation: Implications on self-adapting systems architecture. *Proceedings of Distributed Applications and Interoperable Systems*: *6th IFIP WG 6.1 International Conference*, *DAIS 2006*, Bologna, Italy, 14–16 June 2006. Springer: Berlin, 2006; 64–69.
7. Oreizy P, Gorlick MM, Taylor RN, Heimhigner D, Johnson G, Medvidovic N, Quilici A, Rosenblum DS, Wolf AL. Architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications* 1999; **14**(3): 54–62.
8. Garlan D, Cheng S-W, Huang A-C, Schmerl B, Steenkiste P. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer* 2004; **37**(10):46–54.
9. Lutfiyya H, Molenkamp G, Katchabaw M, Bauer M. Issues in managing soft QoS requirements in distributed systems using a policy-based framework. *Proceedings of 2nd International Workshop on Policies for Distributed Systems and Networks* (*POLICY'01*). Springer: Berlin, 2001; 185–201.

10. Kephart JO, Chess DM. The vision of autonomic computing. *IEEE Computer* 2003; **36**(1):41–52.
11. Walsh WE *et al.* Utility functions in autonomic systems. *Proceedings of International Conference on Autonomic Computing*. IEEE Computer Society Press: Silver Spring, MD, 2004; 70–77.
12. Kephart JO, Das R. Achieving self-management via utility functions. *IEEE Internet Computing* 2007; **11**(1):40–48.
13. McKinley PK, Sadjadi SM, Kasten EP, Cheng BHC. Composing adaptive software. *IEEE Computer* 2004; **37**(7):56–64.
14. Poladian V, Souza J, Garlan D, Shaw M. Dynamic configuration of resource-aware services. *Proceedings of 26th International Conference on Software Engineering*, Edinburgh, U.K. IEEE Computer Society Press: Silver Spring, MD, 2004; xviii+786.
15. Geihs K, Khan MU, Reichle R (eds.). *MADAM Deliverable D3.3*, *UML Modelling Elements and Approach for Application Adaptation*. http://www.intermedia.uio.no/confluence/display/madam/Deliverables [4 August 2008].
16. Dey AK. Understanding and using context. *Personal and Ubiquitous Computing* 2001; **5**(1):4–7.
17. Chen G, Kotz D. A survey of context-aware mobile computing research. *Technical Report TR2000-381*, Department of Computer Science, Dartmouth College, November 2000; 1–16.
18. Mikalsen M, Paspallis N, Floch J, Papadopoulos GA, Ruiz PA. Putting context in context: The role and design of context management in a mobility and adaptation enabling middleware. *International Workshop on Managing Context Information and Semantics in Mobile Environments* (*MCISME*) *in Conjunction with the 7th International Conference on Mobile Data Management* (*MDM*), Nara, Japan, 9–12 May 2006. IEEE Computer Society Press: Silver Spring, MD, 2006; 76–83.
19. Paspallis N, Chimaris A, Papadopoulos GA. Experiences from developing a context management system for an adaptation-enabling middleware. *Seventh IFIP International Conference on Distributed Applications and Interoperable Systems* (*DAIS*) (*Lecture Notes in Computer Science*, vol. 4531), Paphos, Cyprus, 5–8 June 2007. Springer: Berlin, 2007; 225–238.
20. Alia M, Horn G, Eliassen F, Khan MU, Fricke R, Reichle R. A component-based planning framework for adaptive systems. *The 8th International Symposium on Distributed Objects and Applications* (*DOA*), Montpellier, France, 30 October–1 November 2006; 1686–1704.
21. Khan MU, Reichle R, Geihs K. Architectural constraints in the model-driven development of self-adaptive applications. *IEEE Distributed Systems* 2008; **9**(7):13–22. Art. no. 0807-07001.
22. Geihs K, Khan MU, Reichle R, Solberg A, Hallsteinsen S. Modeling of component-based self-adapting context-aware applications for mobile devices. *IFIP Working Conference on Software Engineering Techniques* (*SET*), Warsaw, Poland, 2006; 85–96.
23. XML Metadata Interchange (XMI), v2.1. http://www.omg.org/cgi-bin/doc?formal/2005-09-01 [4 August 2008].
24. MOFScript Model-to-Text Transformation. http://www.eclipse.org/gmt/mofscript/ [4 August 2008].
25. MOF Models to Text Transformation Language, Beta 2, OMG Document Number: ptc/07-08-16. http://www.omg.org/docs/ptc/07-08-16.pdf [4 August 2008].
26. Mascolo C, Capra L, Emmerich W. *Mobile Computing Middeware* (*Advanced Lectures in Network*, *Lecture Notes in Computer Science*, vol. 2497). Springer: Berlin, 2002; 20–58.
27. Aurrecoechea C, Campbell AT, Hauw L. A survey of QoS architectures. *Multimedia Systems* 1998; **6**:138–151.
28. Fitzpatrick T, Blair G, Coulson G, Davies N, Robin P. Supporting adaptive multimedia applications through open bindings. *International Conference on Configurable Distributed Systems* (*ICCDS '98*), Annapolis, MD, May 1998; 128–135.
29. Blair G, Coulson G, Robin P, Papathomas M. An architecture for next generation middleware. *Middleware '98 IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer: Berlin, 1998; 191–206.
30. Kon F, Roman M, Liu P, Mao J, Yamane T, Magalhães LC, Campbell RH. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. *Middleware '2000*, New York, April 2000; 121–143.
31. Katz RH, Brewer EA, Amir E, Balakrishnan H, Fox A, Gribble S, Hodes T, Jiang D, Nguyen GT, Padmanabhan V, Stemm M. The bay area research wireless access network (BARWAN). *COMPCON Spring '96—41st IEEE International Computer Conference*, Santa Clara, CA, 1996; 15.
32. Brewer EA, Katz RH, Chawathe Y, Gribble SD, Hodes T, Nguyen G, Stemm M, Henderson T, Amir E, Balakrishnan H, Fox A, Padmanabhan VN, Seshan S. A network architecture for heterogeneous mobile computing. *IEEE Personal Communications* 1998; **5**(5):8–24.
33. Capra L, Emmerich W, Mascolo C. CARISMA: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering* 2003; **29**(10):929–945.
34. Keeney J, Cahill V. Chisel: A policy-driven, context-aware, dynamic adaptation framework. *Fourth IEEE International Workshop on Policies for Distributed Systems and Networks* (*POLICY 2003*), Lake Como, Italy, 4–6 June 2003; 3–14.
35. Chan ATS, Chuang S-N. MobiPADS: A reflective middleware for context-aware mobile computing. *IEEE Transactions on Software Engineering* (*TSE*) 2003; **29**(12):1072–1085.
36. Noble BD, Satyanarayanan M. Experience with adaptive mobile applications in Odyssey. *Mobile Networks and Applications* 1999; **4**(4):245–254.
37. Heineman GT, Loyall JP, Schantz R. Component technology and QoS management. *International Symposium on Component-based Software Engineering* (*CBSE7*), Edinburgh, Scotland, 2004; 249–263.

38. Sharma PK, Loyall JP, Heineman GT, Schantz RE, Shapiro R, Duzan G. Component-based dynamic QoS adaptations in distributed real-time and embedded systems. *Proceedings of the International Symposium on Distributed Objects and Applications* (*DOA'04*), Larnaca, Cyprus, 2004; 1208–1224.
39. Gjørven E, Eliassen F, Lund K, Eide VSW, Staehli R. Self-adaptive systems: A middleware managed approach. *Proceedings of the 2nd IEEE International Workshop on Self-Managed Networks*, *Systems and Services* (*SelfMan'06*), Dublin, Ireland, 2006; 15–27.
40. Lundesgaard SA, Lund K, Eliassen F. Utilising alternative application configurations in context- and QoS-aware mobile middleware. *Proceedings of the 6th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*. Springer: Berlin, 2006; 228–241.
41. Grace P, Blair G, Samuel S. ReMMoC: A reflective middleware to support mobile client interoperability. *Proceedings of International Symposium on Distributed Objects and Applications* (*Lecture Notes in Computer Science*, vol. 2888). Springer: Berlin, 2003; 1170–1187.
42. Bellavista P, Stefanelli C, Tortonesi M. The ubiquitous provisioning of Internet services to portable devices. *IEEE Pervasive Computing* 2002; **1**(3):81–87.
43. Henricksen K, Indulska J. Developing context-aware pervasive computing applications: Models and approach. *Pervasive and Mobile Computing* 2006; **2**(1):37–64.
44. Henricksen K, Indulska J, Rakotonirainy A. Using context and preferences to implement self-adapting pervasive computing applications. *Software Practice and Experience* 2006; **36**(11–12):1307–1330.