Modeling of Component-Based Self-Adapting Context-Aware Applications for Mobile Devices

Kurt Geihs[1], Mohammad U. Khan[1], Roland Reichle[1]
Arnor Solberg[2], Svein Hallsteinsen[2]

[1] University of Kassel, Wilhelmshoeher Allee 73, FB16,
34121 Kassel, Germany
{geihs, khan, reichle}@vs.uni-kassel.de
http://www.vs.uni-kassel.de/
[2] SINTEF ICT, Strindveien 4,
NO-7465 Trondheim, Norway
{Arnor.Solberg, Svein.Hallsteinsen}@sintef.no

Abstract. A challenge in distributed system design is to cope with the dynamic nature of the execution environment. In this paper, we present a model-driven development approach for adaptive component-based applications running on mobile devices. Context dependencies and adaptation capabilities of applications are modeled in UML. We present our new modeling approach and UML profile. A short description of the required middleware infrastructure is given and the transformation technique of the UML models to platform specific code is briefly introduced. An application example illustrates the modeling and development approach. The presented research results have been obtained as part of the European IST project MADAM.

1 Introduction

Many people carry a mobile device of some sort wherever they go, and an increasingly diverse set of mobile devices (PDAs, smart phones, laptops etc.) are becoming widely available. As a matter of fact, people become more and more accustomed to using mobile services ubiquitously in both work and leisure situations. Clearly, the performance and quality of mobile applications crucially depend on the dynamically changing properties of the execution context, e.g. communication bandwidth fluctuates, error rate changes, battery capacity decreases, and a noisy environment may obliterate the effect of sound output. Therefore, applications on mobile devices need to adapt themselves to their current operational context automatically according to goals and policies specified by the user and/or the developer.

The development of self-adapting applications opens up a great challenge: The range of devices, types of infrastructure, types of context dependencies, ways in which context can change, situations in which users can find themselves and the functions they want, introduce great complexity and demand a systematic, general methodology to design and implement self-adapting applications.

Our goal is to develop such a methodology for future self-adapting applications. We want to provide users with applications that are robust and retain their usability and good performance even in the case of context changes. At the same time we want to free system developers and system managers from much of the low-level details of configuration, operations and maintenance activities.

In this paper we present the modeling of adaptive applications with UML as part of an MDA-based development approach. Self-adapting applications are built as component frameworks with integrated variability, i.e. the application developer specifies variation points when designing an application. During application runtime, if the context changes, adaptation is performed by selecting a suitable application variant, i.e. component configuration, that fits to the current context conditions. All of this is supported by a powerful middleware platform. The choice of using the UML as the modeling language stems from the intention of achieving the benefits of the MDA approach in the application development and complying with popular UML tool environments. Our UML adaptation model is platform independent and it can be automatically transformed to programming language code.

Section 2 of this paper contains an overview of the basic concepts for adaptation and the system architecture and presents the underlying development approach. In Section 3 we present our modeling approach. A new UML profile facilitates the specification of adaptive applications together with its context dependencies. An example is given to illustrate the approach. Section 4 very briefly introduces the model to code transformation technique. Related work is discussed in Section 5. Finally, in Section 6 we comment on experiences with the approach, and we point to future work.

2 Adaptation Approach

The goal of adaptation is to provide the best possible service to the user based on the current context and user preferences. In order to facilitate the development of applications that are able to adapt to context changes, the developer must specify how alternative variants can be derived. These variants provide the same basic functionality but differ in their extra-functional characteristics and resource requirements. In order to facilitate reasoning and decision making about adaptation, the developer must specify the context dependencies and a utility function that is evaluated to estimate the suitability of a variant in a given context. In the following we present a conceptual model for designing applications with built-in variability and, their relation to the context. The underlying middleware that provides the platform to run the applications is briefly introduced.

2.1 Application Variability Model

In our approach applications are component based and the variability is achieved by applying similar concepts as used by the product family community [1]. The variation points are realized by using the concepts of ComponentType and Plan.

Conceptual Component Model. According to Szyperski [2], a Component is a unit of composition with contractually specified interfaces and explicit dependencies where dependencies are specified by stating the required interfaces and the acceptable execution platform(s). Naturally, components may be composed of other software components and may use other components or resources. Our component model is consistent with the concepts of existing component models like CCM, EJB and .NET with the exception that in our current implementation, we use a single class for realizing the atomic component. But this concept is easily extendable.

The conceptual component model of our approach is shown in Figure 1. A Component has exactly one type. A Component Type can have different realizations. It provides and requires services through ports. The characteristics of the component type are defined through its set of Port Types. The Port Type is specified through its set of extra functional characteristics represented by its required and provided QoS_Properties. The functional characteristics of a Port Type are represented by its provided and required interfaces.

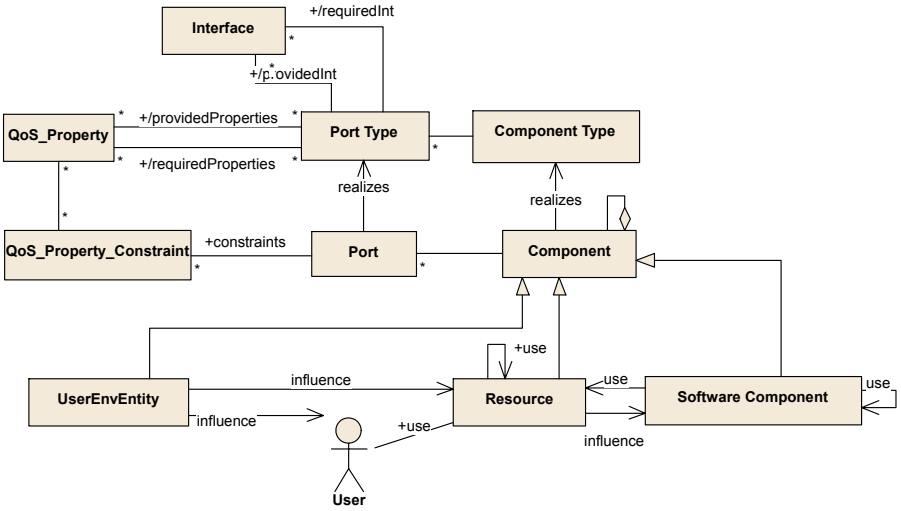

Fig. 1. Conceptual component model

The specification of provided and required services can be seen as defining contracts between components. The Port Types define component contracts through their associated provided and required QoS_Properties. Property constraints associate concrete values with the properties of a component, which may be expressed as constants or as expressions.

A Component can be a SoftwareComponent, a Resource or a UserEnvEntity. A SoftwareComponent can be composed of other components and may collaborate with other software components and use resources through its ports. A Resource represents a run time source of supply and has a limited capacity, which is expressed by its properties. During runtime, consumption and availability of the resource may vary and need to be monitored by middleware services.

The User in the model of Figure 1 represents the actual user of the applications. UserEnvEntity represents entities of interest in the user environment, such as light and noise. The user environment entities may impact resources and other user environment entities.

In our work we have identified three types of context, i.e., user context that relates to the user of a service, system context that encompasses the properties of the execution environment of an application and application context that encompasses the properties of an application providing a service. Context elements are realized through components and context characteristics are expressed as QoS properties of the component.

Dynamic Creation of Application Variants. In our conceptual model an application is represented by a Component Type. The recursive closure of all realizations corresponds to what is often referred to as a component framework [2].

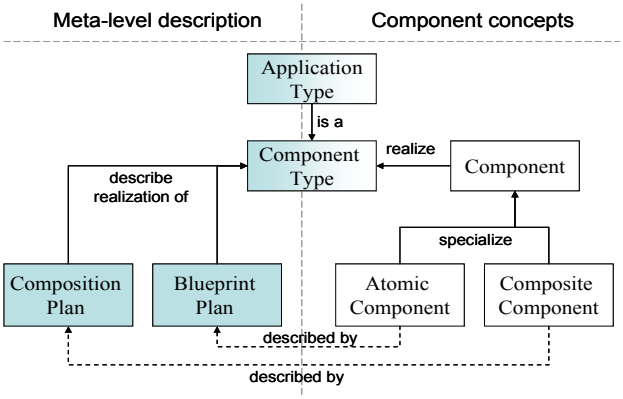

Fig. 2. Component framework

In Figure 2, an application is a software component and an Application Type is considered to be a Component Type. The realization of a Component Type is described using Plans. Components can be atomic as well as composite; accordingly there are two types of Plans: Blueprint Plan and Composition Plan. A Blueprint Plan describes an atomic component and it basically contains a reference to the class that realizes the Component. The Composition Plan recursively describes a composite component by specifying the involved Component Types and the connections between them. A plan represents one possible realization of the associated component type. Variation is obtained by describing a set of possible realizations of a Component Type using Plans.

The representation of applications as Components, Component Types and Plans enables the automatic creation of application variants by recursively resolving the variation points. This reduces the modeling effort significantly. The application developer has to provide only the overall component structure of an application, but there is no need to specify all the possible application variants explicitly. The modeling of one Composition Plan can result in several application variants. In a Composi-

tion Plan only the cooperating Component Types are specified, which in turn can have several different realizations described by their corresponding Plans. Besides, if another component is developed that realizes an already existing Component Type further application variants can be derived without much additional modeling efforts. Only the Blueprint Plan and the extra-functional properties of the component have to be specified.

2.2 Middleware Support for Adaptation

The structure of the middleware platform for running the applications is shown in Figure 3. The Context Manager monitors and processes the context information by means of context sensors. The Adaptation Manager receives context information and decides about the adaptation activities. If adaptation is needed, the Adaptation Manager dictates the Configurator to start up the appropriate configuration.

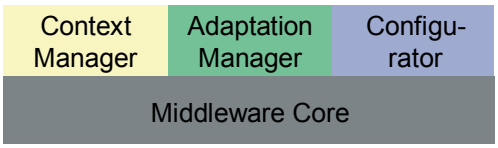

Fig. 3. Middleware building blocks supporting self-adapting applications

The middleware core is responsible for the automatic creation of the application variants as described in the previous section and provides fundamental services for the management of applications, components and component instances. The core relies on the basic mechanisms for instantiation, deployment and communication provided by an underlying distributed computing infrastructure.

The Adaptation Manager reasons on the impact of context changes and is responsible for selecting the application variant (or set of application variants if multiple applications are running) that best fits the current context. In order to evaluate the appropriateness of a particular variant of an application, the utility of the variant is computed. Utility functions along with QoS properties are assigned to each Component Type. For a composite component, the utility value and property constraints can be derived from these.

The Configurator is responsible for the instantiation and configuration of the components that form the selected variants of the running applications.

3 Model Development

We follow the Model Driven Architecture (MDA) approach [3]. An abstract, platform-independent model is needed to capture the adaptation capabilities of complex applications and an automatic transformation to code eases the implementation substantially because it reduces the probability of making mistakes such as omitting possible application configurations in the implementation.

For the platform-independent modeling, we use standard UML 2.0 specification [4]. In addition to this, we extend the standard UML 2.0 specification by introducing a new UML profile, in order to allow generating abstract descriptions of the application's variability and adaptation capabilities. This abstract specification is transformed to appropriate Java code that is responsible for creating the data structures for the Component Types and Plans and for publishing them to the middleware. Our UML-based model builds on experiences with an earlier XML-based model [5].

3.1 UML Profile

We use the UML 2.0 Composite Structure as a baseline in order to model the application architecture. All entities that represent the application context, the resources and the software components (see the conceptual component model in Figure 1) that comprise the applications can be modeled and linked by appropriate associations. For modeling architectural design we have extended the sub-packages InternalStructures and Ports described in the UML 2.0 superstructure specification. The complete description of the profile is beyond the scope of the paper and in the following we present the most relevant part of the profile used for modeling the adaptibilty of applications.

The UML profile defines the component types of the conceptual model by extending the EncapsulatedClassifier of Composite Structures, as shown in Figure 4. Thus, a component type of the conceptual model of Figure 1 is defined as an EncapsulatedClassifier. UserenvEntities, resources and software components are realizations of a Component Type. A Plan describes the realization of a ComponentType.

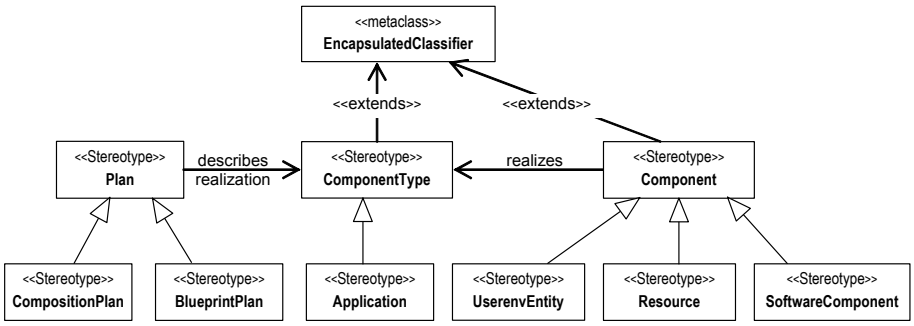

Fig. 4. Plan, Component Type and Component are considered as classes with internal structures

Figure 5 presents the stereotypes for QoS_Property and PortType. The QoS_Property extends the UML metaclass Property in order to express the extra-functional properties of different variants of applications. QoS_Properties are provided or required through ports. Ports realize Port Types. Port Type is an extension of the UML Port metaclass and it is characterized through its provided and required QoS properties.

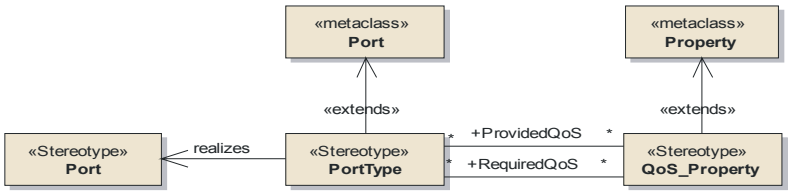

Fig. 5. QoS_Property extends UML Property and PortType is an extension to UML Port meta-classes

For property constraint and utility function we have the following stereotypes: QoS_Property_Constraint, Required_QoS, Provided_QoS and UtilityFunction as shown in Figure 6.

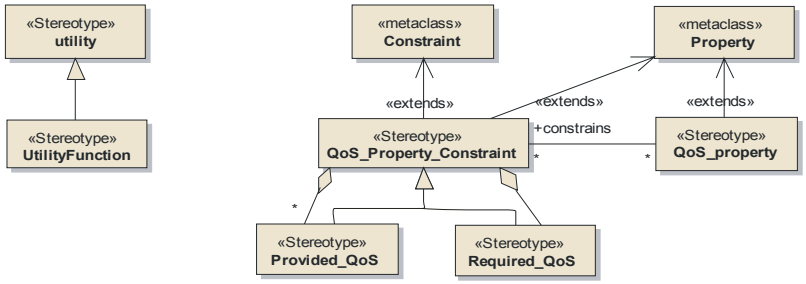

Fig. 6. Property and PropertyConstraint

A QoS_Property_Constraint defines constraints on a Property for a particular component. The Property is provided or required through the port types of the component type. The stereotypes Provided_QoS and Required_QoS include a set of property constraints and indicate if the Property is required or provided respectively. UtilityFunction is a generalization of the UML 2.0 standard stereotype <<utility>>, which designates classes having no instances; but denoting non-member attributes and operations. This provides a basic support that satisfies our requirement of expressing the utility of application variants as functions.

3.2 Modeling Example

The modeling technique has been applied to develop two comprehensive distributed applications. Here we use one of them, namely the SatMotion application in order to illustrate the modeling.

The actual modeling of the application starts with the modeling of requirements along with the context and its resources. Here we present a simplified model to focus mainly on the modeling of self-adaptation. Our emphasis is on the variability model of the application which is used to automatically derive the application variants (architecture). During the adaptation process, the suitability of these variants is evaluated and the best fitting variant is chosen to run.

Description of the Application. SatMotion is a commercial distributed application that facilitates the setting up of Internet connections via satellite terminals (also known as VSAT terminals). It basically provides assistance to the field installer on the antenna alignment procedure. It runs on PDAs (e.g. IPAQ) and conventional laptops. The client software of SatMotion consists of a control module, a command editing module, a graphics module, a math processor module, a recording module and an offline analysis module. The server software consists of a communication control module, a storage module and an instrumentation control module.

The SatMotion application offers two main operating modes: Two-Way and One-Way. For both modes, two sub-modes, BasicClient and Recorder, are available which are active depending on the concrete task to be performed by the user. The Two-Way mode implements a two way communication tool able to command the remote instrument, which receives signal traces information from the server and can also send commands to the server side. SatMotion One-Way is a simplified version of the Two-Way mode, enabling just one-way communication for the reception of information from the server. While the Two-Way mode requires a low-latency and highly reliable connection to perform bidirectional operations, the One-Way mode can work with a lower network quality to offer the same real-time signal visualization to the user. Both operating modes, Two-Way and One-Way are complemented by an offline client mode. This variant needs no network connection and is able to play, perform measurements, generate reports etc. on recorded spectrum activity, received previously in an online mode (either One-Way or Two-Way) and stored on the handheld. As indicated by the three different modes, the self-adaptive capabilities of the application address mainly changes in the network context.

The selection of different variants of the application also depends on the internal and external context and resource conditions of the application. Examples for the resources and context elements are: device resources (power drain, power level, memory, processor), user environment (light source, noise), system infrastructure (I/O extension, screen dimensions, screen colours, brightness), network (type, latency, capacity, throughput), application (status, operating mode), etc.

Variability Model of the Application

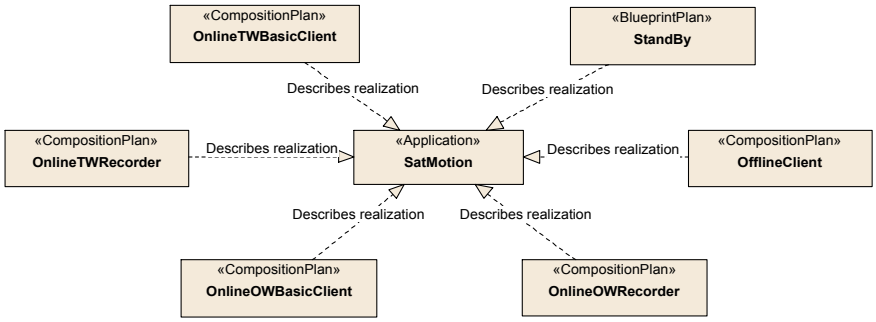

Fig. 7. Plans for the SatMotion application

In the variability model, the SatMotion application is represented as a Component Type that can be realized by any of six Plans, e.g. OfflineClient, OnlineTWRecorder, StandBy etc. as shown in Figure 7. Thus variability is introduced through the possibility of choosing among different plans for the application.

A BluePrintPlan represents the end of the recursion and describes the details of an atomic component. As shown in Figure 7, the SatMotion application can be realized according to one BluePrintPlan and five CompositionPlans. A composition plan is further specified recursively through other composition plans and blueprint plans.

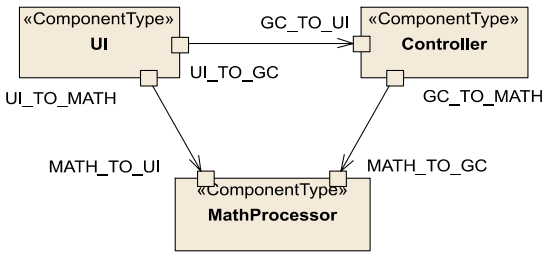

Fig. 8. Component types and their associations in the *OnlineTWBasicClient* Plan

Let us look at the OnlineTWBasicClient plan of Figure 7. Its component composition is shown in Figure 8. The component types *UI, Controller* and *MathProcessor* will be decomposed further until all possible variation points have been resolved and the recursion stops at a BluePrintPlan. Please note that all of these possible variations are evaluated in the Adaptation Manager at run-time in terms of their specified utility.

An example of a BlueprintPlan describing one possible realization of the *Controller* Component Type and providing a *OneWayController* is shown in Figure 9. It contains a definition of a Utility Function, the Component itself and the Property Constraints of the various ports of the component regarding device resources and network.

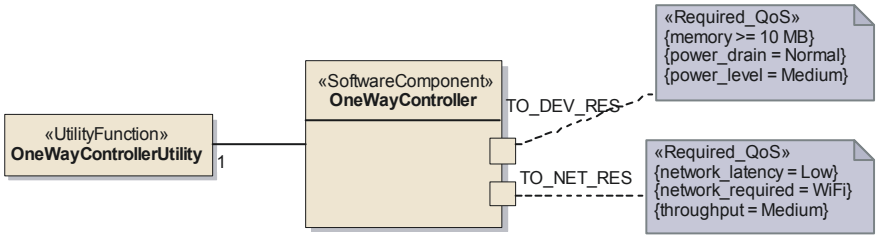

Figure 9: BlueprintPlan for a *OneWayController*

In Figure 9, simple expressions are shown as QoS_Property_Constraints. However, in most cases property constraints can be represented as larger arithmetic expressions involving a number of different QoS_Properties.

It is to note here that there are architectural constraints that can limit the creation of meaningful application variants. For example, a OneWayController can only use a one way mode of the UI, thus its combination with a TwoWayUI will be futile. More-

over, there can be components realized by the same class but with slight changes in their property requirements that can be adjusted by simple configuration parameters. Having separate Blueprint Plans for each of them would cause a big modeling effort. We are currently working on these issues and modeling support will be provided accordingly in the near future.

4 Transformation of the Model to Programming Language Code

Development of the abstract model and performing automatic code generation provides high flexibility. If another target-platform should be addressed, the abstract high level model can be reused, only the transformation has to be adjusted to meet the needs of the new platform. Furthermore, if changes in the overall structure of the applications 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 useful to manage the set of application variants and to ensure completeness.

In our work, we use the Eclipse Modeling Framework, in which UML modeling tools like Omondo and Borland Architect Together can be integrated. The UML model of the application can be exported as XMI according to the metamodel defined by the EclipseUML2, 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, which comes as an Eclipse plug-in. The generated code is then published to the middleware.

The transformation technique is introduced for the completeness of the MDA approach; however, this paper focuses mainly on the modeling aspect and the details of the transformation technique are beyond its scope.

5 Related work

There are several research projects dealing with the development of frameworks and middleware in order to support adaptive applications. Examples are CASA [6], Conductor [7], QuO [8] and Rainbow [9]. In these projects, adaptation modeling mainly focuses on the rules and strategies for adaptation. Coming from a different perspective, TRAP/J [10] supports application adaptation for existing Java applications by means of reflection and aspect oriented programming techniques.

Our work is based on the model-driven development paradigm and aims at platform independent but middleware-specific specifications of the variability and adaptation capabilities of applications. In order to allow the selection of the best fitting application variants based on the utility concept, we have to provide modeling support for the extra-functional properties and property constraints of applications. Therefore parts of our UML profile naturally provide similar modeling constructs as OMG's UML Profile for Quality of Service and Fault Tolerance Aspects [11] which includes modeling support for QoS constraints.

Examples for other research projects exploiting the benefits of the model-driven development paradigm and extending UML for providing the platform independent modeling support are MODA-TEL [12], aiming at the MDA-based development of telecommunication systems, and COMBINE [13], dealing with the component based development of enterprise systems. However, these projects focus on the model-driven development of static applications, whereas we aim at modeling the dynamic variability and self-adaptation capabilities of applications.

Another project developing a framework for adaptive mobile applications and services is FAMOUS [14]. However, the project does not emphasize the model-driven development approach and therefore does not aim at automatic code generation from platform-independent models. In [15] an adaptive middleware framework for context-aware applications is presented. However, it lacks the discussion of the development support for adaptive applications.

6 Conclusions

In this paper we have presented a modeling technique for self-adapting, context-aware applications with UML 2.0 in line with the Model Driven Architecture approach of software development. Our focus has been on the specification of application adaptability. The abstract platform-independent adaptation model is transformed to platform-specific code by a transformation.

The specified modeling and transformation techniques have been applied and tested with the development of two real-life commercial distributed applications. Our experiences are promising and support our initial hypothesis: An abstract, platform-independent model facilitates considerably the engineering of adaptation capabilities of complex distributed applications. The model supports dynamic configuration evaluation and selection of suitable application variants at run-time. The automatic transformation to code eases the implementation to a large extent and it reduces the probability of omitting possible application configurations in the implementation.

While working with the trial applications, we have found out that when adaptation occurs often application configurations are evaluated that are practically infeasible. In order to reduce the computational complexity we need to avoid these configuration plans up-front. As future work, we will extend our modeling support for the concepts like architectural constraints and parameterized components in order to solve the above mentioned problems. The transformation support will be improved as well. We will also generalize our notion of context and adaptation towards service contexts and distributed adaptation scenarios where more than one computing device is involved in the adaptation process.

Acknowledgement

The work presented in this paper is done as part of the MADAM [16] project funded by the European Commission under the 6th framework programme. We would like to

express our thankful gratitude towards the MADAM consortium and the commission for their valuable support.

References

1. Gomaa, H. and M. Hussein (2003), "Dynamic Software Reconfiguration in Software Product Families", 5th Int. Workshop on Product Family Engineering (PFE), Lecture Notes in Computer Science, Springer-Verlag.
2. Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison Wesley, 1997 (2nd ed. 2002, ISBN 0-201-74572-0).
3. OMG MDA Homepage: http://www.omg.org/mda/
4. UML 2.0 Specification: http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf
5. Kurt Geihs, Mohammad Ullah Khan, Roland Reichle, Arnor Solberg, Svein Hallsteinsen, Simon Merral, "Modeling of Component-based Adaptive Distributed Applications" DADS Track, The 21st Annual ACM Symposium on Applied Computing, Dijon, France, April 23 -27, 2006
6. Arun Mukhija and Martin Glinz, "The CASA Approach to Autonomic Applications", Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2005), Paris, France, June-July 2005.
7. Mark Yarvis, Peter Reiher, Gerald J. Popek, "Conductor: A Framework for Distributed Adaptation", Proceedings of the Seventh Workshop on Hot Topics in Operating Systems, 1999.
8. Joseph Loyall, Emerging Trends in Adaptive Middleware and its Application to Distributed Real-time Embedded Systems. Third International Conference on Embedded Software (EMSOFT 2003), Philadelphia, Pennsylvania, October 13-15, 2003.
9. Shang-Wen Cheng, An-Cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste, "Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure", IEEE Computer Vol. 37 Num. 10, October 2004.
10. S. Masoud Sadjadi, Philip K. McKinley, Betty H.C. Cheng, and R.E. Kurt Stirewalt, "TRAP/J: Transparent generation of adaptable Java programs", In Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04), Agia Napa, Cyprus, October 2004.
11. http://www.omg.org/cgi-bin/apps/doc?ptc/04-09-01.pdf
12. A. Gavras, M. Belaunde, L. Ferreira Pires, J.P.A. Almeida. “Towards an MDA-based development methodology for distributed applications.” In: Proceedings of the 1st European Workshop on Model-Driven Architecture with Emphasis on Industrial Applications (MDA-IA 2004), CTIT Technical Report TR-CTIT-04-12, University of Twente, ISSN 1381-3625, Enschede, the Netherlands, March 2004, pp. 71-81.
13. http://www.opengroup.org/combine
14. Hallsteinsen, S., Stav, E. and Floch, J., Self-Adaptation for Everyday Systems, ACM SIGSOFT Workshop on Self-Managed Systems (WOSS '04), Newport Beach, CA, USA, 2004.
15. M. C. Hübscher, J. A. McCann, An adaptive middleware framework for context-aware applications, Personal and Ubiquitous Computing, Vol. 10, No.1, pp. 12–20 (2006).
16. MADAM Project Homepage: http://www.ist-madam.org