Context-Awareness for Self-adaptive Applications in Ubiquitous Computing Environments

Kurt Geihs and Michael Wagner

EECS Department, University of Kassel
Wilhelmshöher Allee 73, 34121 Kassel, Germany
{geihs,wagner}@vs.uni-kassel.de

Abstract. Context-awareness is a prerequisite for self-adaptive applications that are able to react and adapt to their runtime context. We have built and evaluated a comprehensive development framework for context-aware, self-adaptive applications in dynamic ubiquitous computing scenarios. The framework consists of a middleware and an associated model-driven development methodology. In this paper we focus on the context-awareness part of the framework. We discuss design objectives, design decisions, and lessons learnt. The main contributions of this paper are generally applicable insights into the design and seamless integration of context-awareness, dynamic service landscapes, and application adaptation management for applications in highly dynamic execution environments. These insights not only relate to the functional requirements and constraints, but also to non-functional aspects that have a strong influence on the user acceptance of such applications.

Keywords: context-awareness, self-adaptation, middleware, socio-technical requirements.

1 Introduction

Context-awareness is an exciting feature that enables new kinds of self-adaptive applications. Our notion of context agrees with the authors of [17] who stated that *"There is more to context than location"*. Thus, we adopt the more general and widely-accepted definition: *"[Context is] any information that can be used to characterize the situation of an entity"* [1]. Self-adaptive applications monitor and reason about external context conditions and react to changes by automatically adapting their application behavior in order to provide adequate service under changing conditions. Particularly in mobile and ubiquitous computing environments, where dynamic change is a characteristic, often unavoidable feature, self-adaptation is an attractive option, if not a requirement.

Basic hardware and software support for context-aware applications is available today in mobile computing devices such as smartphones and pad/tablet computers that contain sensors for location, acceleration, sound, light, orientation, and more. Needless to say, these mobile computing devices have the computing and memory capacity to execute sophisticated applications and systems software.

Building context-aware and self-adaptive applications is an inherently complex task. Not only need the developers be concerned about the main functionality of the

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2013

application, but also they have to understand which context parameters influence the application functionality, how application variants depend on these parameters, and which variant should be activated under which context conditions. Thus, building such applications requires specific software development and run-time support.

Project MUSIC (Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments) was a European research project in the 6th Framework Program that particularly tackled this challenge. It provided a model-driven development methodology for context-aware, self-adaptive applications, including supporting tools, as well as an adaptation middleware that features sophisticated context management, adaptation reasoning, and application reconfiguration. The results of MUSIC were evaluated through the development of a number of realistic applications and their demonstration in live settings, e.g. in the Paris Métro.

In this paper we focus on the evolution of the context-awareness support in the MUSIC framework, and in particular we focus on insights and lessons learnt that reach beyond MUSIC. These insights were derived as part of the MUSIC activities but also in follow-on projects where the MUSIC technology was employed and evolved. The details of the overall MUSIC approach and technology have been presented already in several publications, e.g. [4, 16]. Consequently, we explain design and implementation details only as far as necessary and refer to more technically focussed papers for further explanations.

The context management middleware of MUSIC was built as a self-contained, separately reusable component. As such it can be used without the adaptation management of MUSIC in order to enable context-awareness features in other application scenarios that do not require adaptation and reconfiguration. Hence, the contributions of this paper are not only relevant within the scope of the MUSIC project, but provide general advice for designers of context-aware systems in highly dynamic application environments.

The rest of this paper is structured as follows: Section 2 covers objectives and requirements for the design of context-aware, user-centric applications in open and dynamic execution environments. In Section 3 we present a very short overview of the MUSIC middleware architecture which is needed for the subsequent discussions. Section 4 discusses the design decisions for our specific context-awareness features that follow from the stated requirements and objectives for dynamic ubiquitous computing scenarios. In Section 5 we present an assessment of these features combined with a comparison to related work. Section 6 concludes the paper and gives an outlook to future work.

2 Objectives and Requirements

Our main goal is to provide context-awareness for self-adaptive applications in mobile and ubiquitous computing environments. Such environments are characterized by continuous change of conditions and by a variable, open service landscape where services may appear and disappear at any time and are provided by independent service providers.

We assume that applications are component-based. Adaptation can be achieved by several adaptation mechanisms, such as compositional (also known as architectural), parametric, distributed deployment, and service-based adaptation. Among these adaptation techniques we have specifically aimed at support for compositional and service-based adaptation: Compositional adaptation allows the modification of the application architecture, i.e. components may be added, removed, or replaced, in order to change the application behavior. Service-based adaptation allows the replacement of an application component by a dynamically discovered and bound external service, if the resulting configuration provides a better utility. The middleware takes care of service discovery, utility evaluation, adaptation planning, and service binding.

Context-awareness is key for all self-adapting systems. As stated above, our notion of context encompasses "*any information that can be used to characterize the situation of an entity*" [1] – as long as there are sensors (i.e. information sources) available for it, one might want to add. Thus, context information includes information on the state of the execution environment (e.g. location, speed, light, sound etc.), the state of the computing device (e.g. computing and memory capacity, battery status etc.) as well as user preferences (e.g. priorities for certain operation modes). In addition to these classical context parameters, the availability of accessible services is considered a special kind of context information. We will come back to this later.

The stated application and context assumptions lead to a number of requirements and challenges for the context management middleware (CMM). First of all, a general software engineering principle needs to be achieved, i.e. *separation of concerns*. This is important in order to clearly separate collecting, storing, providing, and managing context data from the functionality that consumes it. This kind of separation also facilitates the sharing of the context middleware between several applications, i.e. one instance of the middleware can serve several applications concurrently.

Since the CMM is meant to operate in an open dynamic environment sensor availability and sensor heterogeneity are major concerns. If we assume that applications will not only use context sensors that are built into the computing device, but also external sensors, for example an indoor positioning system or an external speedometer, we need a CMM architecture that supports a *flexible configuration* of sensor components and a loose coupling between sensor provider and sensor consumer. Basically, it leads to a "context as a service" model [20] where a sensor is viewed as a dynamically discoverable service providing information on certain attributes of the environment. Thus, language and middleware support is required for offering and requesting context sensors.

Going further along this avenue, the need for mastering the *heterogeneity* of sensor information arises – syntactically as well as semantically. This may involve operations ranging from simple data type conversions (e.g. short to long integer) over translating different metrics (e.g. Fahrenheit to Celsius) to more sophisticated semantic mappings (e.g. street address to GPS coordinates). However, where does the required meta-level information come from? Clearly, some form of semantical representation is required to capture the relationships and potential mappings between data types, parameter metrics, and other kind of context information. A context ontology would be appropriate to store this information. The ontology should also support reasoning about possible transformation chains that convert data items in multiple steps by a set of consecutive intermediate conversions if no direct one-step conversion is available.

Developing context-aware applications requires a well-defined context query interface in order to provide applications with powerful, distribution and device transparent access to different kinds of sensor information. Ideally, the *query language* should support typical sensor related functions such as selecting, filtering, aggregating, and accessing sensor data. While this requirement has been addressed already in several projects that have provided database-like query languages for context management (e.g. [7, 10]), here we emphasize the specific aspects of dynamic and heterogeneous sensor landscapes where designers have to cope with different sensor data types, representations, and semantics.

A key challenge in the envisaged application scenarios is the dynamic appearance and disappearance of services which applications would like to exploit in order to improve their performance. This requires appropriate middleware functionality for discovering services, negotiating service usage agreements, generating service proxies on-the-fly, and binding to services. In addition, in order to integrate the services and their properties into the context-dependent adaptation planning we need to *align the service model with the context model* and relate service properties and context properties. From the perspective of adaptive applications, both kinds of events, i.e. context changes and service appearance / disappearance, may open up opportunities for improving the application utility. Computing the utility of potential application variants generally requires input on the state of the context and the properties of the offered services. Thus, we need an integrated model for an application's context and service dependencies.

Another concern with context-aware applications on mobile computing devices is the *battery capacity*. Generally, smartphone sensors tend to be very resource consuming components. For example, it is well known that continuous GPS tracker applications on smartphones put an enormous stress on the battery. Therefore, from a practical viewpoint we demand that unused context sensors should not consume resources and should be switched off automatically. Likewise, if several applications request the same kind of context data, there should be shared access to the sensor data instead of separate access.

Last but not least there arise non-functional, socio-technical concerns when dealing with context-awareness in user-centric applications. By the term user-centric here we refer to applications that exploit sensitive user data as part of the context-aware behavior. Obviously, tracking, storing, and transmitting data about the activities of a user or even monitoring and processing vital data of a user raises a whole bunch of questions related to security, privacy, legal constraints, trust in the technology, and more. Questions on the legal constraints might be: How can we include legal considerations into the design process such that the processing, storing and sending of user-oriented context-awareness data do not violate existing law? What kind of service contracts do we use (implicitly or explicitly) if third party service providers are involved? Likewise, trust related questions are: How can the user build up trust in a system which monitors the user's context and adapts automatically? Does the system really behave as the user wants it to behave? What kind of technical mechanisms support trust-building of users? How and where are trust-supporting components integrated into UC systems?

The developer of such user-centric context-aware applications necessarily needs to pay attention to these aspects that add further complexity to the development process. Likewise questions related to the usability of the application, i.e. the design of the human-computer-interface, pop up with innovative context-aware and self-adaptive applications: How do we make sure that a user is not overwhelmed by complex context-awareness and self-adaptation features? How can we make sure that the user can handle and interact with a system where many components are hidden in the environment and where many activities happen automatically? What is a good compromise between usage simplicity and attractive functionality? Should we equip these applications with different usage levels for the more or less experienced and skilled user?

Typically, a software engineer will be unable to cope with such non-functional requirements and concerns, and hence interdisciplinary domain experts need to be involved during the requirements, design and evaluation phases. Unfortunately, except for security and privacy concerns, which mostly are viewed as part of the functional requirements, there is little systematic development support available today to ensure that the socio-technical aspects are integral ingredients in the software development process. One of our main research goals is to develop such an interdisciplinary development methodology.

3 Architecture Overview

Before continuing with the discussion of context-awareness and self-adaptation we need to define at least a coarse architectural frame that clarifies our view of the position and role of the context management and adaptation management middleware. Figure 1 illustrates the architecture of the MUSIC middleware which is used as reference architecture for the following discussions. For further details the reader is referred to [16].

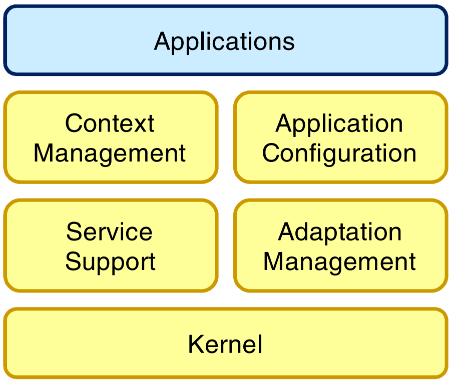

Fig. 1. Basic building blocks of the MUSIC middleware

The middleware implements a control loop which complies with the well-known MAPE (Monitor, Analyse, Plan, Execute) loop in autonomic computing [9]. It monitors the relevant context sensors, and when significant changes are detected, it triggers a planning process to decide if adaptation is necessary. When this is the case, the planning process finds a new configuration that fits the current context better than the one that is currently running, and triggers the adaptation of the running application. To do this the middleware relies on an annotated quality of service-aware architecture model of the application available at runtime, which specifies its adaptation capabilities and its dependencies on context information. This model corresponds to the "Knowledge" component of the autonomic manager in the autonomic computing blueprint. The planning process evaluates the utility of alternative configurations, selects the most suitable one for the current context (i.e. the one with the highest utility for the current context which does not violate any resource constraints) and adapts the application accordingly.

Context information is provided by context sensors and context reasoners in the Context Management middleware which is designed as a separately reusable stand-alone component in the MUSIC middleware. Applications may directly access the context information. The Context Management performs basic reasoning about the type of context changes and their significance for the application. The application designer needs to specify at design time when a context change is considered significant. Furthermore, the application designer needs to provide an adaptation model for the application. The adaptation model specifies all possible application variants and how these variants are related to context parameters. A runtime representation of this adaptation model is stored in the Adaptation Management part of the middleware.

Adaptation planning is triggered by notifications from the Context Management about significant context changes. The Adaptation Management evaluates the utility function for all application variants given the particular context situation, and selects the application variant with the highest utility for the given situation. As a result, the Application Configuration is triggered by the adaptation manager if an application needs to be reconfigured.

A unique feature of the MUSIC framework is the seamless support for discovering and binding external services as part of the context-awareness and self-adaptation. A local software component (e.g. a data storage component) can be replaced by an external service (e.g. a database on a server) if this is a specified application variant and if this configuration leads to a higher application utility. Thus, Service Support in Figure 1 comprises protocols and functions for service discovery, negotiation and monitoring of service level agreements, QoS management, service binding, and more. The Adaptation Management evaluates the available service offerings in terms of their service properties and quality of service guarantees and compares all possible application configuration alternatives when planning an adaptation. Thus, service-based adaptation planning must pay attention to the service availability and depends on the quality of service guarantees that – from the viewpoint of adaptation planning – become a special kind of context parameters.

The Kernel of the middleware provides basic services for communication, storage of metadata, and more.

4 Specific Context-Awareness Features

In this section we return to the requirements of Section 2 and discuss their technical implications and our resulting design decisions.

Separation of Concerns. The MUSIC middleware is built as a collection of clearly separated components that can be evolved and replaced independently. Context Management is decoupled from Adaptation Management, and can be reused without the other middleware components. Context Management is realized as a plug-in architecture where context sensors and context reasoners are plug-ins that can be replaced easily according to the specific application scenarios and requirements. It supports the on-the-fly integration of newly discovered sensors, following a new "sensor as a service" design principle. The activation of the plug-ins is implemented using an automated mechanism which monitors the varying context needs of the applications and starts and/or stops the plug-ins accordingly, thus achieving significant resource savings [11]. While some context plug-ins are readily available in repositories, developers often want to develop their own, tailored to the specific needs of the application and runtime environment. Another benefit of the plug-in concept is that code reuse is greatly facilitated via plug-in repositories, which allow posting, searching and accessing generic as well as specific context plug-ins.

Context Management and Adaptation Management together share the abstract, platform-independent adaptation model that specifies how applications are linked to context parameters and how the values of these parameters affect the utility of the application variants.

Heterogeneity. The targeted application scenarios in mobile and ubiquitous computing environments are inherently open and dynamic. Thus, heterogeneity in many forms is an unavoidable consequence. The foundation for coping with heterogeneous sensor information in the MUSIC framework is a domain ontology that captures the necessary knowledge about mappings between heterogeneous sensor types and data representations [14, 15]. The ontology supports simple data type mappings as well as more elaborate reasonings about the relationship and mapping between different sensor information.

Dynamic Sensor Configurations. As stated above, mobile and ubiquitous computing environments are inherently open and dynamic. This implies not only questions of sensor heterogeneity but also questions of sensor availability, in particular since we must assume that applications may want to access device-external sensors, e.g. for more accurate information. Thus, we have extended the context management middleware of MUSIC towards a loosely coupled "context as a service" model. A sensor is viewed as a dynamically discoverable resource, and language support is provided for offering and requesting context sensors, very similar to general service offer and request languages [20].

Context Query Language. While the Context Management of MUSIC offers a straightforward programmatic API which allows both synchronous context queries (i.e., asking for sensor readings) and asynchronous context queries (i.e., subscribing to context changes by providing a context listener), this simple API has been complemented by a more powerful interface based on a new Context Query Langauge (CQL) in order to support filtering and aggregating context information at a higher level of abstraction in a convenient and transparent way.

For instance, if an application needs the maximum value of some sensor over the last hour, there is no need for the application developer to explicitly access all sensor readings available for the defined time period and infer the desired data from that. MUSIC provides the Context Query Language (CQL) for selecting, filtering and accessing context information [3, 13]. Having such an interface offers two main benefits: First, the developer can work at a higher level of abstraction thus reducing the amount of code and the risk of bugs. Second, the runtime performance is improved because the sensor data is processed while still in the context repository, thus avoiding unnecessary data movement.

CQL is XML-based and allows applications to submit complex queries about an entity or a set of entities of the same type, providing advanced features such as:

- access to current and past context elements - raw or derived from other context data - using a single query;
- filters and constraints to select context data and to subscribe to asynchronous context change events, on both context data and metadata;
- logical operators to combine elementary conditions into more complex ones.

Let's take the user's location as an example. An application can subscribe to periodic notifications (e.g., the position is periodically sent every 10 seconds regardless its specific value), for location changes (e.g. depending on the query, notifications are sent if the geographical coordinates or the civil address change), or for specific location values expressed using query constraints (e.g., notifications are sent only if the user is located in the kitchen). Moreover, the language enables the composition of queries that incorporate semantic references and aggregation (e.g.,"the average passenger age is below 40"). CQL syntax also supports the definition of relations between entities, allowing more abstract queries, such as subscribing to the event "user Mary is at home". More details on the CQL can be found in [3].

Service Discovery as Part of the Context. In order to exploit fully the potential of dynamic ubiquitous computing environments we support the integration of external dynamically discovered services as replacements for local application components if that improves the utility of the application. Thus, the appearance of a new service instance is a context event that may trigger a new round of adaptation planning. If an application component potentially can be replaced by an external service, semantic service discovery and matching are supported by annotating the corresponding component type in the UML-based adaptation model by a specification of the required port types, interfaces, and semantic constraints. Such annotations are attached only to component types that may be the target of dynamic service discovery and binding. Clearly, core components that are crucial for the general functionality of the application will most likely not be candidates for dynamic substitution by externally provided services.

Adaptation reasoning with services depends on the actual service properties, i.e. QoS properties. From the perspective of adaptation reasoning these properties correspond to the context parameters provided by context sensors and reasoners. In order to enable such QoS-driven adaptation reasoning, discoverable services are expected to provide information about their offered QoS properties. In general, QoS guarantees of services are defined in SLAs and established by a negotiation process. Typically, a service can provide different levels of guarantees. Therefore, we have integrated a plug-in negotiation component into the middleware that handles the negotiation of service level agreements. Several standards are available for service level negotiation. The plug-in mechanism of the middleware facilitates the integration of different protocols.

By aligning context model and service model, dynamically discoverable services are made part of the application context that controls the application adaptation.

Distributed Context. An important feature of the MUSIC context management middleware is its support for context distribution, i.e. context information gathered by a node can be shared with other nodes within a defined domain. Distributed context can be used to trigger adaptation on a remote device. For instance, an error occurring on one node may lead to adaptation on other nodes. Context distribution is completely transparent to both the plug-in and application developers. The context middleware keeps track of which devices are available, together with their offered context information types and privacy policies. When a context entity is queried (or subscribed to) the context middleware will know how to fetch the requested data.

Energy Efficiency. In order to save battery power mobile devices cannot afford to run energy hungry resources even if they are not needed currently. Therefore, a local sensor plug-in can be disabled if none of the applications accesses the sensor.

Socio-technical Concerns. Widespread adoption of a new technology, in particular if it is a user-centric technology such as context-aware and adaptive ubiquitous computing, not only depends on the technical progress, but also on "soft factors" that determine the user acceptance. Our goal is to support the development of context-aware applications that are *socially compatible* by design. We intend to avoid the often encountered situation that a new software product is rejected in the end because it has non-technical flaws and risks.

In our research we have asked ourselves what non-functional requirements are crucial to the acceptance, i.e. social embedding, of user-centered context-aware UC applications, and we decided to concentrate first on three key concerns: trust, usability, and legal conformance. Clearly, there are more than these three areas of socio-technical concerns in the development of user-centered context-aware applications. This is future work. Note that indeed we view security and privacy of user data as very important concerns in ubiquitous computing applications. From our perspective these elementary concerns are part of the technical requirements. Therefore, we explicitly include security and privacy provisions in the technical requirements of an application and not in the socio-technical requirements.

Socio-technical requirements are difficult to assess and to translate into technical artefacts in the middleware or application. For example, there is no prefabricated component that encapsulates the handling of trust issues inside a modular component. In our solution the socio-technical concerns trust, usability, and legal conformance are taken into account in the design process of a context-aware application, particularly in the requirements analysis and conceptual design phases.

5 Assessment and Related Work

The MUSIC framework including its context management middleware was evaluated by building and experimenting with a range of prototype applications in real application environments, such as live experiments in Metro stations in Paris. A detailed discussion of evaluations and measurement results can be found in [2]. Here we focus particularly on the discussion of the design of context-awareness features.

The main trade-off we faced during the design of the overall context management system, and particularly the context modelling, was that of sophistication versus simplicity. On the one hand, we aimed at a context management approach that included state-of-the-art practices for context gathering, modelling, and management. But at the same time we also aimed at a flexible and extensible infrastructure that would allow us to replace functionality as needed by different application scenarios and thus to facilitate the experimental evaluations, e.g., replacing the inter-representation transformations or the context repository.

The design of the overall context management approach, the context model, and the corresponding context query language [13, 14] was heavily influenced by previous work on context-aware systems [6, 7, 8, 18]. However, none of the existing solutions provided all the features that we needed for ubiquitous, self-adaptive computing applications in open and dynamic environments. So we had to select and integrate features from several best-of-breed state of the art solutions. For example, our basic information model for context sources is based on [18]: Every context entity is associated with a scope and a representation. Mapping of different scopes and representations in a heterogeneous environment can be performed if the context ontology contains the required semantic relationships. Another example for how the MUSIC context management reuses and extends existing solutions is our two-level domain ontology that stands behind the context model. It is similar to the one of SOCAM [5]. However, we distinguish between a top-level ontology capturing general concepts and global knowledge and domain-specific lower level extensions for application specific details. In particular, we extend the SOCAM approach by provisions for the service-based adaptation, described in the previous section.

A strong and unique feature of our context model is the ability to model nearly any type of context information via an extensible ontology, including service interfaces and service properties. Furthermore, the context query language excels at raising the abstraction level of the way developers specify context information and context filtering conditions, while the context middleware automatically handles the representation heterogeneity and transformations, and thus takes this burden from the developers. Context model, context query language, and context ontology go hand in hand in a synergetic manner.

This level of abstraction and transparency comes at a price. Design, implementation, and configuration of the context management system have been rather complex undertakings, which is mainly due to the richness of features and capabilities. On the other hand, extensive experience with building context-aware applications on top of the MUSIC context management middleware has revealed that some of the features are rarely used. As mentioned above, the context management approach is built on top of a context ontology that enables sophisticated functionality including automatic transformation of context data between different representations, inference of context information based on the relationship of the corresponding entities, as well as semantic disambiguation. This rich functionality is blamed by application developers as a source of relatively high cost in terms of learning and application development effort. Application developers have particularly pointed to the steep learning curve of understanding and using the context model [12]. There is room for improvement and fine-tuning in future versions.

6 Conclusions

The overall goal of the MUSIC project was to facilitate the engineering of self-adaptive applications in ubiquitous computing environments, which are characterized by inherent openness, dynamism, and heterogeneity. The project has delivered a comprehensive development methodology and middleware framework for self-adaptive applications. The technical results were tested and evaluated by building a range of application demonstrators. The experiments proved the viability and effectiveness of the MUSIC achievements.

In this paper we have focused on the context management middleware of MUSIC. We have highlighted the unique requirements and resulting features of context management in dynamic and open ubiquitous computing application scenarios where dynamically discoverable services are considered part of the application context. In particular, our intention was to convey experience with context-awareness that have not been documented elsewhere. Thus, the main contributions of this paper are advice and guidelines for developers of context-aware systems.

Overall MUSIC has opened up several new avenues for further research that are being tackled in follow-on projects. For example, the interdisciplinary VENUS project [19] at the University of Kassel aims at an enhanced development methodology for self-adaptive ubiquitous computing applications that explicitly includes extra-functional concerns, such as usability of adaptive software, trust in context-awareness features, as well as legal constraints on the gathering, storing, and processing of personal context information. We believe that convincing answers to such questions are crucial for the acceptance of new technologies, in particular if they involve user-related context information.

Acknowledgments. The contributions of the MUSIC and VENUS project teams are gratefully acknowledged.

References

1. Dey, A.K.: Providing architectural support for building context-aware applications. PhD thesis, College of Computing, Georgia Institute of Technology (2000)
2. Floch, J., Frà, C., Fricke, R., Geihs, K., Wagner, M., Lorenzo, J., Soladana, E., Mehlhase, S., Paspallis, N., Rahnama, H., Ruiz, P.A., Scholz, U.: Playing MUSIC — building context-aware and self-adaptive mobile applications, Software: Practice and Experience. John Wiley & Sons, Ltd. (2012), doi:10.1002/spe.2116
3. Frà, C., Valla, M., Paspallis, N.: High level context query processing: an experience report. In: Proceedings of the 8th IEEE Workshop on Context Modeling and Reasoning (CoMoRea 2011) in Conjunction with the 9th IEEE International Conference on Pervasive Computing and Communication (PerCom), pp. 421–426. IEEE Computer Society (2011)
4. Geihs, K., Reichle, R., Wagner, M., Khan, M.U.: Modeling of Context-Aware Self-Adaptive Applications in Ubiquitous and Service-Oriented Environments. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 146–163. Springer, Heidelberg (2009)
5. Gu, T., Wang, X.H., Pung, H.K., Zhang, D.Q.: An Ontology-based Context Model in Intelligent Environments. In: Proc. of Communication Networks and Distributed Systems Modeling and Simulation Conference, pp. 270–275 (2004)
6. Henricksen, K., Indulska, J.: A Software Engineering Framework for Context-Aware Pervasive Computing. In: IEEE Int. Conf. on Pervasive Computing and Communications, pp. 77–86 (2004)
7. Henricksen, K., Indulska, J.: Developing context-aware pervasive computing applications: Models and approach. J. of Pervasive and Mobile Computing 2(1), 37–64 (2006)
8. Hönle, N., Käppeler, U., Nicklas, D., Schwarz, T.: Benefits Of Integrating Meta Data Into A Context Model. In: Proc. of IEEE PerCom Workshop on Context Modeling and Reasoning, pp. 25–29 (2005)
9. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
10. Korpipää, P., Mäntyjärvi, J., Kela, J., Keränen, H., Malm, E.J.: Managing context information in mobile devices. IEEE Pervasive Computing 2.3, 42–51 (2003)
11. Paspallis, N., Rouvoy, R., Barone, P., Papadopoulos, G.A., Eliassen, F., Mamelli, A.: A Pluggable and Reconfigurable Architecture for a Context-Aware Enabling Middleware System. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part I. LNCS, vol. 5331, pp. 553–570. Springer, Heidelberg (2008)
12. Paspallis, N.: Middleware-based development of context-aware applications with reusable components, PhD thesis, University of Cyprus, Nicosia, Cyprus (2009)
13. Reichle, R., Wagner, M., Khan, M., Geihs, K., Valla, M., Fra, C., Paspallis, N., Papadopoulos, G.A.: A Context Query Language for Pervasive Computing Environments. In: IEEE Int. Conf. on Pervasive Computing and Communication, pp. 434–440 (2008)
14. Reichle, R., Wagner, M., Khan, M.U., Geihs, K., Lorenzo, J., Valla, M., Fra, C., Paspallis, N., Papadopoulos, G.A.: A Comprehensive Context Modeling Framework for Pervasive Computing Systems. In: Meier, R., Terzis, S. (eds.) DAIS 2008. LNCS, vol. 5053, pp. 281–295. Springer, Heidelberg (2008)
15. Reichle, R.: Information Exchange and Fusion in Dynamic and Heterogeneous Distributed Environments, PhD thesis, University of Kassel, Kassel, Germany (2010)

16. Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., Scholz, U.: MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 164–182. Springer, Heidelberg (2009)
17. Schmidt, A., Beigl, M., Gellersen, H.-W.: There is more to Context than Location. Computers & Graphics Journal 23(6), 893–901 (1999)
18. Strang, T., Linnhoff-Popien, C., Frank, K.: CoOL: A Context Ontology Language to Enable Contextual Interoperability. In: Stefani, J.-B., Demeure, I., Zhang, J. (eds.) DAIS 2003. LNCS, vol. 2893, pp. 236–247. Springer, Heidelberg (2003)
19. VENUS Project, `http://www.iteg.uni-kassel.de/venus/`
20. Wagner, M., Reichle, R., Geihs, K.: Context as a service - Requirements, design and middleware support. In: Proceedings of the 9th Annual IEEE International Conference on Pervasive Computing and Communications, PerCom 2011, Seattle, WA, USA, March 21-25, pp. 220–225. IEEE (2011)