Infrastructural support for dynamic context bindings

Tom Broens, Aart van Halteren and Marten van Sinderen

Centre for Telematics and Information Technology, ASNA group,
University of Twente, P.O. Box 217, 7500 AE Enschede, the Netherlands,
{t.h.f.broens, a.t.vanhalteren, m.j.vansinderen}@utwente.nl

Abstract. Research in context-aware systems shows that using context information enables the development of personalized mobile applications. The context acquisition process in a context-aware (CA) system consists of two main roles: context producing entities (e.g. wrapped sensors) and context consuming entities (e.g. CA application). A CA system can be seen as a hierarchy of associated context producers and consumers which exchange contextual information. Managing contextual information used in context aware systems, introduces additional complexity for mobile application developers. We focus on the dynamic processes of discovery, selection, (re)binding and monitoring of entities that produce context information. Dynamic context binding is complex because of the dynamic nature, in terms of availability and quality, of context producers. We propose to delegate the responsibility for context binding to the middleware infrastructure and provide application designers with a declarative language to specify context information requirements on a high-level of abstraction. In this way, our Context-Aware Component Infrastructure (CACI) provides support for dynamic context bindings between application components and context producers.

1 Introduction

Ubiquitous computing envisions computer systems everywhere, which aid users in a tailored and unobtrusive manner [1]. Context-awareness is a major enabler of this paradigm and offers promising ways to adapt and personalize (mobile) applications. Context-aware (CA) systems take besides explicit user input also the context of an entity (i.e. person, place or object relevant to the functioning of the system) into account, to provide functionality which is adapted to the users situation [2]. This is particularly interesting for mobile applications because these application function in constantly changing environments due to the movement of the user [3]. For example, a mobile tourist guide application could benefit from context by offering personalized tourist information based on the current physical location of the user [4].

Main elements in CA systems are software entities that produce or consume contextual information. Typical *context producers* are wrapped sensors that acquire context from the physical environment (e.g. GPS, temperature sensor, ECG sensor). Typical *context consumers* are CA applications that use context from context producers to adapt their functionality. Some software entities are both consumer and producer. For

example context reasoners, which produce derived context information based on context received from other context producers. A CA system can be seen as a hierarchy of associated context producers and consumers which exchange contextual information (see Figure 1). The association between a context consumer and a context producer is called a *context binding*. Exchanging contextual information consists then of two phases:

1. Create and maintain a context binding between a context consumer and producer (this includes discovery, selection and binding to a context producer by the context consumer).
2. Exchange of contextual information (this includes using a suitable context exchange protocol and context data format).

Fig. 1. Example hierarchy of context producers and consumers in a CA system.

Due to missing infrastructural support, developers of first generation CA systems (e.g. [5]) programmed bindings between context consumers and context producers in an ad-hoc and tightly-coupled fashion, unique for a specific application [2, 6]. The developers choose specific context producers (e.g. GPS location sensors, RFID sensors) and program the low-level interaction between the specific context producer and his application (context consumer). Thereby, he creates a tight coupling between his application and the used context producers. Reuse of the created application is limited and future evolutions (e.g. upcoming of new technology) becomes difficult [2, 7].

Currently, there is trend towards middleware infrastructures for CA systems [8]. These infrastructures offer solutions to recurring problems in the CA domain, like context discovery, reasoning, adaptation and security. Using the run-time discovery and binding mechanisms these infrastructures offer, context producers and context consumers are decoupled and can be bound at run-time. However, establishing and maintaining the context binding is not trivial and still needs extensive programming effort [9]. Developers need to still create programming code to discover, select and bind to relevant context producers for every context consuming entity in the system. Furthermore, due to the mobility of the user, or possibly the context producer, the availability of the context producer for the context consumer is not guaranteed and reliable [10]. Maintaining the binding is therefore complex and needs additional programming effort to develop a flexible and robust context-aware system that can handle this dynamicity.

In this paper, we propose to shift the responsibilities of establishing and maintaining context bindings to the infrastructure and offer a generic binding transparency to application developers of CA systems. With our infrastructure (called CACI), we offer

a binding specification language which enables developers to specify required bindings on a higher level of abstraction. Using this specification, application developers define the context information requirements in stead of specifying the specific context producer they need. Furthermore, CACI provides mechanisms to transform the binding specification from the developer into dynamic bindings which react to changes in availability of context producers. Infrastructural support for dynamic context bindings decreases programming effort for binding and the general complexity of CA applications. Furthermore, it provides a generic approach which is suitable for different types of CA system, thereby stimulating reuse. Additionally, it is a way to get a separation of concerns of the non-functional requirements of context binding and the functional requirements of the CA system. This leaves the application developer with more room to concentrate on the CA application logic. Summarizing, the goal of CACI is to facilitate application developers in quickly and easily developing a flexible and robust CA system.

The remainder of this paper is structured as follows: section 2 presents two real-life scenarios which give further motivation for an infrastructure supporting dynamic context bindings. Section 3 discusses an analysis of CA systems, providing the foundation for our infrastructure. Section 4 provides background information on component-based application development used in our infrastructure. Section 5 discusses the design of the CACI infrastructure, including CA component model, component description language and the CACI binding mechanisms. Section 6 presents related work and finally section 7 gives conclusions and directions for future work.

2 Scenarios

The following two scenarios indicate possible real-life applications of CA systems. The first scenario discusses a CA office environment while the second presents a CA healthcare environment.

2.1 CA office environment

Jerry works at a large bank. He is an account manager which is often preparing for meetings. He is looking for his colleague John, to give him a report for his next appointment. Jerry's CA system indicates that John's location is unknown. A moment later, John walks into the office building and his system logs into the CA office system. At that moment Jerry's system notifies him that John is walking towards the canteen. John can now intercept Jerry and transfer the document.

2.2 CA healthcare environment

Sophie is an epileptic patient suffering from regular seizures. Currently, she wears an epilepsy safety system (ESS) that enables her to be mobile with a feeling of safety. Sophie is walking to the groceries store and the ESS detects a likely occurrence of a seizure. Sophie is warned to sit down and relax while the health care center can notify a nearby and available care giver (e.g. family doctor, voluntary care giver, ambulance

personal) of her state. The selected care giver can proceed to her location to provide first aid [11].

2.3 Discussion

Context-aware systems acquire and process contextual information. This requires a binding between context consuming applications and context producers. In the case of the office scenario, Jerry's system needs location of John. In the case of ESS, Sophie's and the healthcare system need the location of Sophie and the care givers, and the availability of care givers. Without a supporting infrastructure the developer of these applications needs to implement low-level interactions with specific context producers for every piece of context information. Current CA infrastructures support developers by offering discovery mechanisms and context exchange mechanism. However the dynamic properties of context bindings are neglected. For instance, in the office environment Jerry's system needs to get John's location (from a context producer that provides John's location) which is at that time not available. It is just not known which context producer will pop-up or leave at what time. To create a flexible and robust system, developers have to cope with this scenario and need to program some kind of complex monitoring strategy (when supporting functions are at all offered by the infrastructure) that incorporates all foreseen situations. Similarly in the healthcare environment, Sophie is mobile and moves between administrative domains that each exports all kind of different context producers with different properties (e.g. quality of context information). Based on the available context producers the right one has to be bound and this binding has to be monitored. For example, Sophie walks past an electronic shop that reads her RFID tag and can determine that she is in front of the electronic shop. This context producer provides Sophie's location to her. When she moves out of the domain of that store this context producer will disappear and another location producer has to be bound. Without a supporting infrastructure, developers need to program this complex binding strategy (discovery, selection, (re)binding and monitoring). We propose to relieve application developers from this task and to shift this responsibility to the infrastructure using high-level context binding specifications.

3 Analysis of context-aware systems

Characteristics of context-aware systems and context producers influence the binding process. Therefore they need to be taken into account in our context binding mechanism.

3.1 Context and Context-Awareness

The most used definition of context is provided by Dey [2]. He defines *context* as any information that can be used to characterize the situation of an entity. An entity can be a human, a physical or a computational object. Common categorizations of context are: computing context (e.g. email received, bandwidth available), social context (e.g. health, mood, calendar) and physical context (e.g. location, temperature) [12]. *Context-awareness* is defined as a property of a system that uses context to provide rele-

vant information and/or service to the user, where relevancy depends on the user's task.

Context consists of several layers of abstraction (see Figure 2, based on [13]). First, context is always related to an *entity*, which can be a person or an object. Without this entity information, context has no meaning [14]. Then context consists of the real contextual information which is made up out of the context *element* (e.g. location), *value* (e.g. 50.234, 6.152) and *format* (e.g. lat/long). Finally, context has meta properties like *security* policies (e.g. who may read this context) and *quality of context* (e.g. accuracy) [15].

Fig. 2. Context model.

3.2 Context producer properties

Context producers have some properties which indicate the main challenges our context binding transparency addresses:

- *Distributed:* context is offered by a multitude of physically distributed context producers. Problems that arise are how-to discover relevant context sources and how-to exchange context information. Current CA infrastructures focus mainly on this aspect. We position our infrastructure as an extension of current CA infrastructures and leverage on their facilities.
- *Dynamic availability:* the visibility of context producers for context consumers is subject to change. They can appear and disappear at arbitrary moment. For instance, when a user moves out a certain domain the context sources of that domain will disappear (see Figure 3). The availability of context producers is therefore not guaranteed and reliable.
- *Dynamic quality of context:* Furthermore, the quality of the produced context, i.e. Quality of Context (QoC) [15], can vary among context producers and also among context samples provided by a single context producer. Therefore, providing high quality CA applications requires incorporating dynamic quality aspects.

- *Heterogeneous context models:* (similar) context can be provided by different context producers using different data models for storing, accessing and transferring contextual information.

This paper focuses on overcoming the indicated dynamicity challenges (availability and QoC) of context producers. We consider the heterogeneity aspect as a future extension.

Fig. 3. Dynamic availability of context producers.

4 Context-Aware component-based application development

A common type of infrastructure is middleware, which gives some advantages for the development of CA systems. CACI applies the component-based middleware paradigm.

4.1 Middleware and CA systems

Middleware is often characterized as a software layer between applications and the underlying hardware platform. Main goals of middleware are [16]:

- Provide interoperability between distributed applications across heterogeneous platforms;
- Offer programming abstractions that hide complexities of building a distributed application. These abstractions are called transparencies [17] (e.g. the location transparency hides the physical location of the application instead of using a logical name or unique identifier to contact the distributed application).
- Offer common building block that solves recurring problems and thereby facilitate the development of distributed applications.

Middleware technologies have evolved, amongst others, from Procedural, Transactional, Message-oriented, Object-Oriented towards Component-based middleware. In general, we saw this evolution in middleware mechanisms to be able to manage the increasingly complexity of distributed applications. Context-awareness adds another layer of complexity to distributed applications (see section 3.2). To fully enable the ubiquitous computing vision, CA systems should provide a loose coupling between context producers and context consumer. A CA application (context consumer) should

be able to use arbitrary context producers based on their availability and offered characteristics.

In general, middleware infrastructure approaches are beneficial for context-aware systems. If we consider the basic features of middleware infrastructures they facilitate the challenges that developing context-aware systems pose:

- Context producers and context consumers are distributed on possible heterogeneous platforms. Middleware infrastructures can facilitate interoperability between these entities.
- As indicated, developing CA systems is complex. Therefore, there is a need for common abstractions that hide the complexity of specific aspects (e.g. binding, secure access) in the development of CA systems.
- Recurring problems like context discovery, security of context information, context-based adaptation and binding can be bundled into generic middleware building blocks.

4.2 Component-based middleware for CA systems

Component-based middleware views an application as a composition of components. Szyperiski [18] defines a *component* as: “a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties”. Figure 4 indicates the generic architecture of component-based middleware.

Fig. 4. Component-based middleware.

Components are encapsulated by a container that offers the execution environment for components. They are deployed in the container using a component descriptor. Components interact with the container using the container API. The container (and also components) interacts with clients using some type of communication mechanism offered by the lower-level middleware. Generally, lower-level middleware infrastructures offer generic middleware services like transaction support, security, persistency and notification. The component offers certain functionality specified in interfaces to clients. The client can instantiate, destroy and find components using the home interface. The client receives a reference (or handle) to the component on which it can invoke remote method calls. Examples of currently available component-based middleware technologies are Corba Components, J2ME, J2EE and OSGi.

Traditional non-component-based applications can be modeled as a function that transforms an explicit user input into an output. With CA this model is extended with

a context input offered by some context producer. If we then zoom into an application and apply the component-based paradigm, a CA application becomes a composition of CA components. The component can have individual context requirement and context offerings (see figure 5).

Fig. 5. From monolithic context-unaware applications to component-based CA applications.

Current context-aware infrastructures are mainly OO-based solutions. We claim that a direction towards component-based middleware approaches is beneficial for the easy development of context-aware applications because:

- *Reusability of components:* this is a common advantage of components. They are well-defined encapsulated units of programming that can be easily re-used.
- *Third party composition:* When developing a CA system, context consumers and context producers are subject to third party composition. Components are well suited for this third party composition.
- *Unit of deployment:* components execute in a run-time environment (often called container) this means that on deploy-time this environment can execute certain functionality. For CA systems this could include initializing the context bindings and setting security policies.

5 CACI architecture and design

CACI offers a component-based approach to context-aware applications (see Figure 6). It considers a context-aware application as a composition of context-aware components. These components are deployed in a component environment called the container. The boundaries of a component are described using application ports and context ports. Application ports (like in CCM [19]) are descriptions of the functionalities a component requires and can offer. Generally, this is the functional interface of a component. In CACI, we take a similar approach for context ports. Context ports define the context requirements or offerings a component has. These ports are de-

scribed in the component descriptor which is used at deploy-time to configure the container. Configuration actions the CACI binding mechanism takes based on the descriptor are:

- Discovery and initial binding with required context producers.
- Initializing the dynamic re-binding monitoring process.

Fig. 6. CACI component model.

The remainder of this section discusses CACI in further detail. It starts with a description of the CACI component description language. Then, it presents details on the design of CACI binding mechanism. Finally, it discusses the internal design of a CACI CA component.

5.1 CACI component description language

The CACI Component Description Language (CCDL) describes the context requirements of a CACI component. The description is the knowledge base for deployment and operational management of this component and its binding. Figure 7 presents the meta-model of CCDL expressed in UML.

Fig. 7. Meta-model CCDL

CCDL describes, besides ‘standard’ application ports, context binding specifications (i.e. context ports) relevant to the component. A context binding specifies which context is required or offered. Currently we only consider context requirements, facilitating the development of context producers by offering a context offering specification is considered future work. Furthermore, the specification specifies to which entity this context belongs and which quality parameters it has or should have. Furthermore, it specifies the *scope* of the binding. Scope describes if the binding can only be to context sources within the container (local) or also from the environment (global).

Additional parameters for the context bindings are:

- *Priority*: indicates the order to resolve the binding.
- *Policy*: can be dynamic, semi-dynamic or static. Dynamic indicates that the binding has to be monitored at run-time and when ‘better’ context producers pop-up, a re-binding process has to start. Additionally, when the already bound context producers leave, a new context producer has to be bound. Semi-dynamic indicates that only when an already bound context producer leaves, a new context producer has to be bound. Static indicates that binding will be done once at deploy-time.
- *Optional*: is a Boolean value which indicates if the deployment of the component should fail when a binding cannot be resolved.

Figure 8 gives examples of CCDL binding specifications in pseudo-code, useful for the two scenario’s sketched in section 2. CCDL is still under research. The information model, for instance how-to define QoC criteria, is subject of future research.

```
CA_office_env_context_binding ::
  requires Location
  from Colleague.John
  expressed_in LatLong
  with accuracy > 75%
  scope global
  policy semi-static

CA_healthcare_env_context_binding ::
  requires Availability
  from Doctor.Smith
  expressed_in Boolean
  with accuracy > 95%
  scope global
  policy dynamic
```

Fig. 8. Example of a CCDL context binding specification.

5.2 CACI binding mechanism

Figure 9, gives an overview of the deployment and operational phase of a CACI CA component. When a component is deployed, its CCDL descriptor is parsed by the deployer. The deployer sends a binding request to the context binder which generates and instantiates a context producer proxy. It discovers other local container context sources and when appropriate (scope is global), it sends a request to the underlying

Fig. 10. Binding algorithm

5.3 CACI internal design

A major goal of CACI is to offer a binding transparency which has the features of ODP location and relocation transparencies [17]. To offer these transparencies, CACI applies the proxy pattern [20] for its internal representation of a deployed CACI CA component (see Figure 11). The proxy pattern shields the CACI context consuming component from the remote context producer. It creates an intermediary proxy component. This proxy is a representative of the context producing component and performs processing needed for maintaining the binding.

Fig. 11. Appliance of the proxy pattern in CACI.

We support two common types of context acquisition strategies:

- *Request-Response*: the context consuming component explicitly requests for a context sample from the context producer.
- *Subscribe-Notify*: the context consuming component subscribes to context updates triggered by the context producer.

6 Related work

Several context-aware middleware infrastructures [8, 21-23] offer context management functionality. Generally, this functionality enables context consumers to discover and bind to context producers using programming statements. Often dynamic monitoring capabilities (e.g. notification when new and possibly better, context sources arise) are not available and when context producers become (un)available the decision to re-binding and the choice to bind to which context source has to be taken by the application rather than the infrastructure.

Cervantas [24] proposes a component model that enables autonomous adaptation of the binding of component services at run-time, using a component description language. These principles are implemented in OSGi as the Service binder. Furthermore, this service is the foundation for the OSGi Declarative service, adopted in release 4 of the OSGi specification [25, 26]. The service binder solely considers components, which may require or offer services, within the scope of one OSGi container, thereby limiting the scope of binding to a single computing system. Furthermore, the scope of the service binder is generic services. Therefore the capabilities of the specification language and its corresponding mechanisms are not tailored to context specific needs. This implies that specific aspects of context like QoC are not supported. QoC highly influence the binding process and therefore needs a central place. However, we adopted some of the aspects supported in their component description language.

Bottaro [27] extends the Service binder principles towards distributed services. Furthermore, dynamic re-binding based on service availability is supported. The binding choice is not specified in the component specification but has to be programmed. Context information can be retrieved and stored (on registration) as properties which can be used by the binding decision. However, dynamic context quality changes that trigger rebinding, throughout the lifetime of the service, are not incorporated.

7 Conclusions and future work

Context is important for ubiquitous environments that want to offer personalized applications in an unobtrusive way. Context producers are the providers of context information. Binding to context producer is challenging because of their distributed, heterogeneous and dynamic nature. These context sources offer the information for context-aware applications to adapt their behavior.

In this paper we introduce CACI which is an application infrastructure simplifying the development of context-aware applications. CACI is based on the component-based middleware paradigm which generally improves reusability and flexibility compared to traditional OO- and RPC-based middleware paradigms. Specifically, CACI introduces a context-aware component model which defines that a component besides application ports also consists of context ports. These context ports specify the context requirements and the context offerings of the component. Especially the context requirements can be influenced by context meta-properties like QoC.

Further, CACI offers a component description language that can be used to declaratively specify a component's context requirements and context offerings. This specification is used by the CACI infrastructure to resolve at run-time the context requirements or registering the context offerings to the global environment, depending on the component's role as context consumer or context producer, respectively. This context description is a natural extension of already available component descriptions. It decreases programming code and the general complexity of the component. It is also a way to get a separation of concerns of the non-functional requirements of context binding and the functional requirements of the component.

With CACI, the binding between context producing and context consuming components is flexible. When context producers/consumers leave or enter the system the bindings are re-evaluated and when suitable a seamless rebinding process is started.

We implemented a proof-of-concept prototype based on the OSGi component framework [28] using the Oscar implementation [29]. The OSGi framework offers lifecycle management of components (called bundles). This includes installation, starting, stopping and updating of components. We reuse the powerful life-cycle management facilities of OSGi and extend it with CACI's context-aware component mechanisms. This extension consists of deploying the CACI binding functionality as a standard component in the container. A CACI component adds a pointer to a CCDL description in its component descriptor (i.e. manifest in the jar file of bundle). At deploy time the context manager parses the CCDL description and establishes bindings. At runtime these binding are monitored. When context sources pop-up or leave, the bindings are evaluated and when necessary a re-binding process is initiated. CACI runs on mobile or fixed system capable of running a java J2ME PP1.0 virtual machine (i.e. in case of windows mobile, IBM J9 is tested). Although CACI is currently implemented on top of OSGi, its principles are not specific for this technology and will be validated in the future on other platforms.

In this paper we present our efforts toward an infrastructure supporting the easy and rapid development of CA applications. However several aspects remain for future research:

- *Support for developing context producers:* currently CACI focuses on easily establishing context bindings for context consumers. However similarly, we could support the developers of context producers by letting them specify their context offerings in CCDL. CACI can then do the registration to the environment and tackle issues like access control to a producer and graceful degradation of a context producer in case of a malfunctioning sensor.
- *Binding algorithm:* further research is needed in the monitoring and re-binding mechanism of CACI. Questions that arise are how-to sense a lost binding, which policy is needed to make a decision on rebinding and how-to do this seamless re-binding.
- *Formalization of CCDL:* the basic structure of CCDL is presented in this paper. However, the information model supporting CCDL needs further research.
- *Stability:* dynamic re-binding has advantages but could also influence the stability of the system. Research is needed on how-to avoid oscillating bindings between context consumers and producer.

- *Proof-of-concept*: To indicate the generality of the CACI approach we plan to deploy CACI on other lower-level middleware technologies like the Jini Service Container [30].

Acknowledgement

This work is part of the Freeband AWARENESS Project . Freeband is sponsored by the Dutch government under contract BSIK 03025. (<http://awareness.freeband.nl>)

References

1. Weiser, M. and J. Brown, *The Coming Age of Calm Technology*. 1996.
2. Dey, A., *Providing Architectural Support for Context-Aware applications*. 2000, Georgia Institute of Technology.
3. Satyanarayanan, M., *Fundamental Challenges in Mobile Computing*, in *Fifteenth ACM Symposium on Principles of Distributed Computing*: Philadelphia, USA.
4. Pokraev, S., et al., *Service Platform for Rapid Development and Deployment of Context-Aware, Mobile Applications*, in *International Conference on Webservices (ICWS'05), Industry track*. 2005: Orlando, Florida, USA.
5. Korkea-aho, M., *Context-Aware Applications Survey*, 2000, Available from: <http://users.tkk.fi/~mkorkeaa/doc/context-aware.html>.
6. Pascoe, J., *Context-aware software*. 2001, University of Kent: Canterbury.
7. Ebling, M., et al., *Issues for Context Services for Pervasive Computing*, in *Workshop on Middleware for Mobile Computing*. 2001: Heidelberg, Germany.
8. Henriksen, K., et al., *Middleware for Distributed Context-Aware Systems*, in *DOA 2005*. 2005, Springer Verlag: Agia Napa, Cyprus.
9. Banavar, G. and A. Bernstein, *Software infrastructure and design challenges for ubiquitous computing applications*. *Communications of the ACM*, 2002. **45**(12): p. 92-96.
10. Bellavista, P., et al., *Dynamic Binding in Mobile Applications*. *IEEE Internet Computing*, 2003. **March-April**: p. 34-42.
11. Broens, T., et al., *Towards an application framework for context-aware m-health applications*, in *EUNICE: Networked Applications (EUNICE'05)*. 2005, ISBN: 84-89315-43-4: Madrid, Spain.
12. Schilit, B., N. Adams, and R. Want, *Context-Aware Computing Applications*, in *IEEE Workshop on Mobile Computing Systems and Applications*. 1994: Santa Cruz, CA, USA.
13. Broens, T. and A. Halteren, *SimuContext: Simply Simulate Context*, in *International Conference on Autonomic and Autonomous Systems (ICAS'06)*. 2006: Silicon Valley, USA.
14. Dockhorn Costa, P., et al., *Towards Conceptual Foundations for Context-Aware Applications*, in *Third International Workshop on Modeling and Retrieval of Context (MRC'06)*. 2006: Boston, USA.
15. Bucholz, T., A. Kupper, and M. Schiffers, *Quality of Context: What It Is And Why We Need It*, in *Workshop of the HP OpenView University Association 2003 (HPOVUA 2003)*. 2003: Geneva.

16. Emmerich, W., *Software Engineering and Middleware: A Roadmap*, in *22th ICSE 2000*. 2000: Limerick, Ireland.
17. Blair, G. and J. Stefani, *Open Distributed Processing and Multimedia*. 1998: Addison-Wesley.
18. Szyperski, C., D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*. Component Software serie, ed. C. Szyperski. 2002: Addison-Wesley.
19. Wang, N., D. Schmidt, and O. O'Ryan, *Overview of the CORBA Component Model*, in *Component-based software engineering: putting the pieces together*. 2001. p. 557-571.
20. Buschmann, F., et al., *Pattern-oriented software architecture: a system of patterns*. 1996: Wiley.
21. Kranenburg, H. and H. Eertink, *Processing Heterogeneous Context Information*, in *Next Generation IP-based Service Platforms for Future Mobile Systems workshop*. 2005: Trento, Italy.
22. Kummerfeld, B., et al., *Merino: Towards an intelligent environment architecture for multi-granularity context description*, in *workshop on User Modelling for Ubiquitous Computing*. 2003.
23. Bardram, J., *The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications*, in *Pervasive Computing*. 2005: Munchen, Germany.
24. Cervantas, H. and R. Hall, *Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model*, in *26st International Conference on Software Engineering*. 2004: Edinburgh, Scotland.
25. OSGi Alliance, *OSGi Service Platform Core Specification: Release 4*. 2005.
26. OSGi Alliance, *OSGi Service Platform Service Compendium: Release 4*, 2005.
27. Bottaro, A. and A. Gerodolle, *Extended Service Binder: Dynamic Service Availability Management in Ambient Intelligence*, in *International Workshop on Future Research Challenges for Software and Services (FRCSS'06)*. 2006: Vienna, Austria.
28. OSGi Alliance, *The OSGi Service Platform - Dynamic services for networked devices*, 2005, Available from: <http://osgi.org>.
29. Oscar.org, *Oscar - An OSGi framework implementation*, 2005, Available from: <http://oscar.objectweb.org/>.
30. Cheiron, *Cheiron project site*, Visited 2006, Available from: <http://www.cheiron.org/>.