A Survey of Context Adaptation in Autonomic Computing

Cornel Klein, Reiner Schmid
Siemens AG – Corporate Research & Technologies
Otto-Hahn-Ring 6, 81730 Munich, Germany
Email: {cornel.klein|reiner.schmid}@siemens.com

Christian Leuxner, Wassiou Sitou, Bernd Spanfelner
Technische Universität München, Department of Informatics
Boltzmannstr. 3, 85748 Garching, Germany
Email: {leuxner|sitou|spanfeln}@in.tum.de

***Abstract*—Autonomic Computing (AC) is an emerging paradigm aiming at simplifying the administration of complex computer systems. Efforts required to deploy and maintain complex systems are usually high. Autonomic Computing may help to reduce these efforts by allowing administrators to define abstract policies and then enable systems to configure, optimize and maintain themselves according to the specified policies. Context adaptation can be regarded as an enabling technology for future applications in the field of Autonomic Computing. In this paper we present a survey of past and future secrets of this enabling technology in Autonomic Computing.**

Keywords: context adaptation, autonomic computing, self-healing, self-organization, self-optimization, self-protection.

I. Introduction

Autonomic Computing aims at simplifying the administration of complex computer systems. The four principal characteristics of AC are self-healing, self-organization, self-optimization and self-protection. These four attributes are often referred to as the four *self-X* paradigms of AC. According to these paradigms, system administrators merely specify high level policies, which determine how the system may adjust its behavior at runtime in order to guarantee specified requirements. Administrators are consequently relieved from dealing with numerous details of the system. While this goal is absolutely desirable still many questions remain open [1] when considering AC Systems.

We consider most AC aspects to be closely related to context adaptation. Context adaptation is a system's capability of gathering information about the domain it shares an interface with, evaluating this information and changing its observable behavior according to the current situation [2]–[4]. A high level policy specified to enable the system's adjustment also can be regarded as context – as well as any other aspect of the system's environment. The individual capabilities of autonomic systems (i.e. self-healing, self-organization, self-optimization and self-protection) require the system to infer from the current situation as well as the policy, and to trigger an adequate measure. Therefore context adaptation is probably what AC systems basically do.

To promote AC we argue for using approved methods and techniques from the context adaptation domain in order to enrich the development of autonomic applications. In the reminder of this paper we take a look at the pervasion of context adaptation in AC by discussing specific requirements to AC applications. Afterwards we present the results of our investigations in evaluating architectural frameworks appropriate for the AC domain. Thereafter some future considerations regarding dynamic quality of service to enhance AC applications are presented and illustrated by means of a hospital scenario. To enable a better understanding of our investigations, we narrow these considerations down to the self-healing attribute, although the major part of the results may also apply to the other self-X attributes.

II. Running Example

In the following, we make use of a scenario from a hospital setting. In this scenario, a network management system manages the following video services:

- Remote Expert: a physician is hooked up by video during a surgery to assist in difficult tasks.
- Local Expert: a local physician is requested by another hospital to be hooked up to a surgery.
- Video Security: several cameras record sensitive parts of the hospital. The images are shown to a guard via monitor and in addition are recorded to a recording system in the central of the security service outside the hospital.
- IPTV: patients are provided with internet-TV and further movies from a movie server.

The four services are connected to a broadband network. The network is fast enough to enable parallel execution of all these services with high quality. Within the scenario suddenly a failure in a central optical network component occurs, resulting in a reduced network bandwidth provided by the backup system (SDSL network). Although the services are reasonably prioritized, it may be necessary to run some of them in parallel – even if this can not be accomplished with the available network bandwidth. In such cases the AC subsystem tries to *dynamically* prioritize over all services currently demanding network bandwidth, thereby allocating the remaining bandwidth appropriately to at least a subset of the currently active network services.

III. Specific Requirements for Autonomic Computing

Like in any other application domain, Autonomic Computing applications require some common properties apart from the generic self-X attributes. These properties are often interrelated and contribute in their combination to a proper achievement of the AC paradigms. In the following we summarize and discuss these properties, denoted as specific requirements for AC.

Adaptability

The core concept behind adaptability is the general ability to change a system's observable behavior, structure or realization [3], [5]. This requirement is amplified by *automatic* adaptation. Automatic adaptability enables a system to decide about an adaptation by *itself* – in contrast to an ordinary adaptation, which in turn is decided and triggered by the system's *environment* (e.g. a user or administrator). Adaptation may affect the change of some functionality, algorithm or parameters as well as the total system structure or any other aspect of the system. If an adaptation comprise the change of the complete system model, including the the model that actually decides on the adaptation and deploys the decisions, this system is called a total reconfigurable system. In case a change of behavior can be expressed by exchanging some functional entities, the system is simply called reconfigurable. Automatic adaptation requires a model of the system's environment. This model is often referred to as *context*. Thus, automatic adaptation is often called context adaptation. We use adaptation and context adaptation synonymously for convenience.

In the hospital scenario from section II, the context adaptation inter alia comprises the exchange of broadband for SDSL network and vice versa. In addition, management functionalities for managing the prioritization, QoS and (de-)activation of the video services are needed.

Awareness

Awareness is closely related to adaptation and context, as it is a prerequisite for automatic adaptation. It has two aspects: self-awareness enabling a system to observe its own system model, state, etc. and awareness of the environment. As stated above, the model of the system's environment is often called context [5] [6]. To be more precise, we denote context as the sufficiently exact characterization of the situations of a system by means of perceivable information that is relevant for the adaptation of the system [3]. In principle any (measurable) characteristic of the system or its environment could be considered for adaptation decisions. A systematic to model a system's context is proposed in [4]. The system in our running example is however only aware of the available network bandwidth and the requested services.

Monitoring

Since monitoring is often regarded as a prerequisite for error discovery and handling, it constitutes a subset of awareness [7] [8]. The peculiarities of monitoring are not discussed within this document. However the notion of monitoring is relevant for this discussion, since it is closely related to the notion of context. Context embraces the system state, its environment, and any information relevant for the adaptation. Consequently, it is also a matter of context, which information for instance indicates an erroneous system state and hence characterizes a situation in which a certain adaptation is necessary. In this case, adaptation can be compared to error handling, as it transfers the system from an erroneous (unwanted) system state to a well-defined (wanted) system state. The monitored context within the running example comprises the network availability, its bandwidth and the services' requests for bandwidth together with their priority.

Dynamicity

Dynamicity embraces a system's ability to change during runtime. In contrast to adaptability this only constitutes the technical facility of change. While adaptability refers to the conceptual change of certain system aspects, which does not necessarily imply the change of components or services, dynamicity is about the technical ability to remove, add or exchange services and components. Once more, there is a close but not dependable relation between both dynamicity and adaptation. Dynamicity may also include a system's ability to exchange certain (defective or obsolete) components without changing the observable behavior. Dynamicity deals with concerns like preserving states during functionality exchange, starting and stopping functionality etc. The dynamicity aspect of the running example manifests in the priority management, which activates and exchanges network components.

Autonomy

As the term Autonomic Computing already suggests, autonomy is one of the essential characteristics of such systems. AC aims at unburdening human administrators from complex tasks, which typically requires a lot of decision making without human intervention (and thus without direct human interaction). Autonomy however is not only intelligent behavior but also an organizational manner. Context adaptation is not possible without a certain degree of autonomy. A rule engine obeying a predefined set of conditional statements (e.g. *if_then_else*) is the simplest form of autonomy. In many cases, such a simple rule based mechanism however may not suffice. The example scenario facilitates *force feedback learning* and *learning by observation* to refine the decisions concerning the priority of services and their granted QoS, respectively.

Robustness

Robustness is a requirement that is claimed for almost every system. AC application will specially benefit from robustness since this may facilitate the design of system parts that deal with self-healing and self-defense. In addition the system architecture could ease the appliance of measures in cases of errors and attacks. Robustness states the first and most obvious step on the road to dependable systems. Beside a special focus on error avoidance, several requirements aiming at correcting errors are forced. Robustness is often achieved by decoupling and asynchronous communication. Both are approved techniques in software and systems engineering, which help in preventing from error propagation.

Mobility

Mobility enfolds all parts of the system: from mobility of code on the lowest granularity level via mobility of services or components up to mobility of devices or even mobility of the overall system [8], [9]. Mobility enables dynamical discovery and usage of new resources, recovery of crucial functionalities etc. Often mobile devices are used for detection and analysis of problems. In the running example, mobility of code is used to transfer some functionalities relevant for security recordings, from the outsourced security central to a local resource, in order to save bandwidth and to facilitate temporally local recordings.

Traceability

Traceability enables the unambiguous mapping of the logical onto the physical system architecture, which inter alia facilitates an easy deployment of necessary measures [8]. The notion of traceability is once more closely related to that of adaptation: adaptation decisions namely also base on an abstract system model in order to reduce the necessary computational power. These decisions are afterwards deployed in the physical system, too. The deployment is usually automatic, and thus requires traceability. Traceability is additionally helpful when analyzing the reasons for wrong decisions made by the system. Traceability aspects are for instance concerned within the running example, whenever a certain decision pertaining the presence or availability of a service is specified on an abstract level (e.g. in an abstract policy), and this very decision can be mapped onto a command implementing this decision, e.g. by (de-)activating or exchanging the corresponding service.

IV. Evaluation of Architectural Frameworks

There is a number of architectural styles and frameworks that already support at least some of the above mentioned requirements for Autonomic Computing applications. In the following, a brief characterization of currently available styles and frameworks is given in order to enable a later mapping onto the AC requirements. We evaluate the frameworks especially regarding their suitability to develop AC applications exposing especially the self-healing characteristic. However, the other self-X attributes are not fully ignored.

Accord Framework

The accord framework [10] was designed to cope with three challenges: heterogeneity, dynamism and uncertainty. As the inventors had a focus on grid environments, these three problems are especially immanent. Therefore the framework implements three requirements that aim at solving these problems:

1) separate interface definition
2) separate computational behavior from interaction and coordination
3) computation, interaction and coordination should be context aware to adapt them to "dynamic requirements"

1) and 2) are attributes of component and service based architectures. 3) is introduced to cope with dynamism. Accord does not explicitly include the application model into the context. The accord framework can be classified as a managed agent based architecture. Components act as agents but are observed and managed by a controller. Replacing components is quite complex, since components maintain a state, which has to be migrated during the replacement.

Weaves Framework

In Weaves [11], messages are objects that are forwarded and manipulated throughout the system. Weaves facilitates blind communication: connectors and components are separated, components do not know sources or destinations of objects and neither their semantics. This ensures that components can be replaced without rearranging connections and vice versa. Weaves architectures can be edited on the fly. As weaves uses asynchronous communication, typical problems arising from connection loss that are problematic for synchronous communication, are avoided.

C2 Framework

C2 [12] is another architectural style that focuses on a hierarchical organization of components to enable decoupling. Components are connected via connectors and are only aware of components that are above them. Therefore direct invocation can only be made from the bottom to the top. Communication is asynchronous and based on a request/reply pattern. However state changes can be propagated via the connectors top-down. Therefore components on a lower level can be exchanged without causing problems on an upper level.

PitM Framework

PitM [13] is an extension of the C2 style aiming at "Programming in the Many". Inter-components connection is extended in a way, that – in addition to top and bottom connectors – side connectors enable synchronous component interaction. As a constraint, two components may not be in a side-to-side and top-down connection at the same time. This restriction prevents from ports misuse. Behavior is described by means of provided and required services and interaction via event-based communication. Furthermore, special connectors called border connectors abstract from distribution over devices, since components may not see the device borders. PitM has a second architecture level called meta-level. Components on this level act as effectors that are aware of the application-level components and may interact with them. This meta-level controls the application behavior. Application data messages are used simply for application related communication, whereas component content, architectural model and system monitoring messages are used to coordinate adaptive features.

CAWAR Framework

The CAWAR framework [14] [15] [9] [3] differs from the former described approaches in that it is a purely service based approach. CAWAR distinguishes four basic service types:

sensors acquire data, *interpreters* process data and *actuators* deploy instructions. *Context elements* are abstract and possibly distributed information buffers, which decouple the other three service types dealing with the data processing. Certain constraints govern the possibilities for composition and ensure unambiguous data processing. The usage of services abstracts from implementation details and focuses on describing the behavior of the specified system. All CAWAR architectures are self-describing, i.e. the system model itself is part of the system and stored in a dedicated context element. This enables inferencing from the model. The CAWAR architecture contains a special management service called *model activator*. This service implements the adaptation decisions within the system model, which may lead to a reconfiguration of the system. Since the activator itself is part of this model, even the activator may be affected by such an adaptation decisions (total reconfigurability). The communication between context elements and the other service types is synchronous, while it is asynchronous in-between the three other types due to the mentioned decoupling via context. Beside the service communications, direct communications on the component layer are also possible and can be negotiated via the context. The context thus buffers information concerning the system states[1], the system model itself and the system environment.

Summary of the Evaluation

Each of the architectural styles described above supports a subset of the former described requirements. All of them are suitable for designing self-healing application, or more generally AC application with specified self-X attributes. However, some architectural styles better support the design of certain self-X requirements than others. Table I summarizes the evaluation results, whereas "o" means the architecture style does not explicitly support the requirement; however it does not avoid its fulfillment in principle. "+" means the architecture style somehow supports the requirement, while "++" expresses, that the corresponding requirement is highly supported by the architecture. As mentioned in the introduction, we focus our considerations on the self-healing attribute, although support of the further attributes may be straightforward.

	Accord	**Weaves**	**C2**	**PitM**	**CAWAR**
Adaptability	+	o	o	++	++
Awareness	+	+	+	+	++
Monitoring	+	o	o	++	++
Dynamicity	+	+	+	+	++
Autonomy	+	+	+	+	+
Robustness	+	+	+	+	+
Mobility	o	+	+	++	++
Traceability	o	o	o	+	++

TABLE I
EVALUATION OF DIFFERENT ARCHITECTURE STYLES

We consider both PitM and CAWAR as promising approaches for equipping system with self-X characteristics, since they support the design of context adaptive systems. C2 and Weaves state first attempts in supporting more flexible system designs. Accord has a slightly different focus, since it is designed for grid environments, where context is usually limited to indicate the availability of certain resources within the computer grid.

Both PitM and CAWAR support all requirements for Autonomic Computing. However, autonomy for any style and framework is highly dependent on the applied techniques for decision making. CAWAR provides a more sophisticated awareness concept than PitM, since CAWAR architectures dictate an explicit model of the system environment in form of context. Moreover, CAWAR inherently supports self-awareness of the system model for inference purposes, whereas PitM merely considers awareness of application related aspects; adaptation related aspects of the system are not taken into account. CAWAR explicitly integrates adaptation and application aspects: the context adaptive system behavior is specified by only four service types, thus enabling a comfortable design of system architectures. Thanks to management services like the model activator, this architecture can be discovered, deployed and reconfigured at runtime.

[1]the involved services are usually stateless.

V. QUALITY OF SERVICE IN AUTONOMIC COMPUTING

An objective of the self-healing paradigm of AC is to support an continuous maintenance of crucial system functionality. In several systems certain system functions are less important than others. Since resources are typically very limited (e.g. due to failures), a compromise between function availability/service provisioning and the quality of the corresponding service is necessary. Crucial services are consequently maintained, while less important services suffer from quality reductions or deactivations.

In terms of our hospital scenario this means the following: due to a failure in a network component, the available bandwidth is restricted. When a remote expert for assisting a surgery is requested (crucial service), the quality of less important active services (e.g. video security) is automatically reduced. This requires a mechanism capable of changing both the required quality of a service and the offered quality of a component (e.g. network bandwidth). Another illustrative example is the selection of an appropriate sorting algorithm: commonly a fast algorithm like "MergeSort" (or an equivalent) is deployed for solving a given sorting problem. Suddenly a failure occurs, which heavily restricts the memory resources. As a self-healing measure, the system exchanges the fast but memory greedy algorithm for a slower algorithm like "BubbleSort", which however merely requires linear memory space. With this scenario in mind, we introduce the concept of Dynamic Quality of Service (Dynamic QoS).

A. Basis of Dynamic QoS Modeling

If systems should expose Dynamic QoS characteristics, it is crucial to enable decision-making concerning the QoS aspects. The technical details of this decision-making are out of the scope of this document. However, it should be mentioned, that fuzzy decision-making algorithms like Bayesian networks or neural networks may often facilitate the design of the required decision logic.

As common to service based approaches, a service model (S-Model) represents a management layer on which such decisions can be made[2]. We will therefore adhere to the S-Model notion as supported for instance by both the PitM and CAWAR architecture [13] [14]. However, any model that allows for a dynamic exchange of services and service-fulfilling components can be used – as long as the system has the ability to access its own model.

Changing the QoS of a given service can be understood as a substitution of services. The process for changing the QoS of services is therefore very similar to that of any other reconfiguration: once the system model is adapted (e.g. modify QoS of some service), an adequate component offering the desired functionality with the specified QoS is searched. The searched component need not inevitably be a different component. If the old component is able to offer the new QoS, it remains a candidate for the service provisioning. However, describing the QoS change via a reconfiguration allows for using QoS beyond the capabilities of the former used component. A detailed explanation will follow after the general mechanism for changing QoS is introduced.

For enabling a system to change QoS aspects via the adaptation mechanism, the system must be self-aware, i.e. it must access a representation of its own model description. The QoS parameters are usually specified within this system model. Hence, an obsolete service description within the model is substituted for a new description containing the desired QoS parameters. When the changed model description is deployed by some management service, the system is reconfigured to contain an adequate component for the changed service description.

Technically, the reconfiguration could be realized as follows: the decision-making entity computes a new system model (with appropriate QoS) by using any relevant information available as context – in particular the current system model. The changed system model is read by the entity that implements the system model at runtime, i.e. by the model activator. The model activator notices all changes within the system model. It tries to discover appropriate components, that fulfill the service policies within the model description – including the corresponding QoS parameters. Sophisticated components may also expose several QoS levels (e.g. low, average, high). As soon as all necessary components are discovered, the activator binds each service contained in the system model to the component fulfilling the specified service with the desired QoS.

The described process looks quite complex for reconfiguring a single service-fulfilling component in order to offer another QoS. However, we prefer this mechanism instead of just sending a configuration message to the concerned component for the following reason: the described mechanism works for all kinds of components. Simple components exposing at most one QoS level as well as components offering multiple QoS levels can be reconfigured by that mechanism. The decision-making procedure thereby also considers information concerning the runtime detection of components and other available resources. Finally, the whole decision process remains reconfigurable itself.

[2]and in fact, it also models the information these decisions are based on.

B. Application of Dynamic QoS for the Hospital Scenario

We now illustrate the reconfiguration mechanism described in the previous section by means of our running example. We consider a scenario, in which QoS parameters of certain video security services are reduced and IPTV is deactivated. These reconfigurations are necessary in order to provide the required network bandwidth for an important incoming video service (e.g. remote expert). We will use the graphical notation of CAWAR for representing the considered system model.

In the initial service model (fig. 1), the occurrence of a failure in the network is recognized by a dedicated sensor called *NetworkBlackoutSensor*. The information concerning this failure is stored within the system context. The interpreter *NetworkSwitchController* decides on basis of this context, that the Broadband Network (component), currently fulfilling the *Network* service, must be replaced by a SDSL network (component). As a result, the initial system model is adapted: the model description specifying the *Network* service is adjusted to reflect the changed bandwidth restrictions (depicted by a dashed line in figure 1). In the simplest case a reference is added to the service description of *Network*, which indicates the new component fulfilling this service, i.e. the SDSL network. The *Model Activator* (not shown in the figure) reads and implements the modified model description by binding all specified services to available components (reconfiguration).

In the mean time the *NetworkScheduler* and its predecessors are activated by the same reconfiguration (not depicted). Now the *AvailableBandwidthSensor* senses the available bandwidth (further sensors for priority requests etc. are omitted). The *NetworkScheduler* modifies the model description for the different video services by adjusting their QoS parameters or even ordering their deactivation to release the allocated network bandwidth for the important video service.

Although a change of QoS does facilitate the same mechanisms there is a slight difference. We do not provide a distinct service model for each possible QoS assignment, and we also do not assume a calculus for model alteration. We merely require the decision-making interpreter (e.g. *NetworkScheduler*) to know the exact peculiarity of the QoS parameters of the affected service (e.g. runtime, fidelity etc.).

Within our scenario setting, we assume that a remote expert is requested. Therefore the quality of the video security shall be reduced. It may not be deactivated due to legal constraints (e.g. observation of rooms where medicines are stored) but a change from a 1024x768 resolution with 25 fps to a 640x400 resolution with 15 fps would release enough network bandwidth for the remote expert video service. The reduced resolution suffice to recognize unauthorized persons

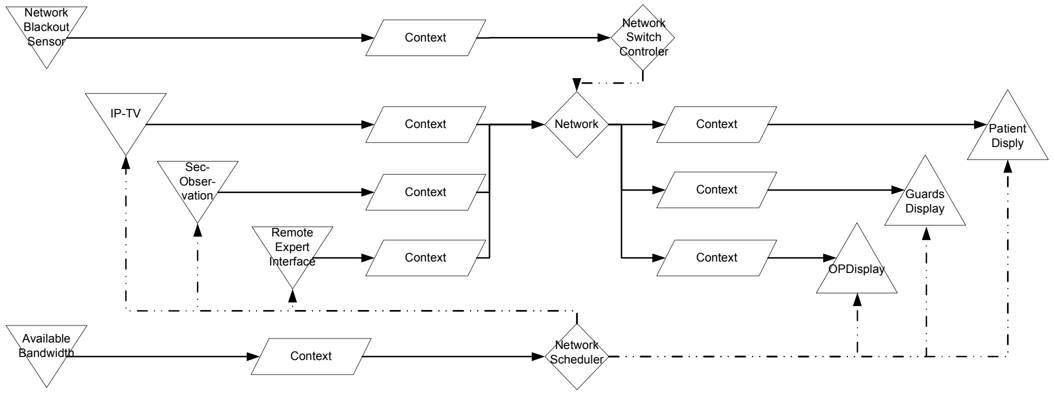

Fig. 1. System model used for demonstrating dynamic QoS

and to trigger countermeasures, although a precise identification may not be possible. The changed QoS in the service model is interpreted by the model activator and finally results in changed service-component bindings. Assuming that the security-camera offers different image resolutions, it is bound and configured accordingly by the model activator.

VI. Conclusion and Outlook

We presented a survey of existing work on context adaptation within the domain of Autonomic Computing. We analyzed the requirements for context adaptation in AC and evaluated some approaches concerning these requirements. It turned out, that a couple of approaches already fulfill many of these requirements satisfactorily. We illustrated how adaptation can be used to enrich the fulfillment of the self-healing requirement. Dynamic QoS has been used to facilitate graceful degradation and the optimal sharing of resources, respectively. The concepts were demonstrated using a hospital scenario, in which several video services were considered.

We recommend context adaptation as a promising approach for realizing the visions of AC, including the support of all self-X attributes. While several ongoing work attempts to address some of the AC paradigms (self-healing, self-organization, self-optimization and self-protection), there has been little work on an integrated approach that takes the entire development process into account. Software engineering constitutes a holistic approach for developing systems that fulfill their requirements. But there is no conclusive modus operandi for developing neither context-aware systems, nor Autonomic Computing systems by now. Thus, further effort is directed towards the elaboration of a continuous approach for developing such systems.

References

[1] J. O. Kephart, “Research challenges of autonomic computing,” in *ICSE ’05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM, 2005, pp. 15–22.

[2] D. M. Berry, B. H. Cheng, and J. Zhang, “The Four Levels of Requirements Engineering for and in Dynamic Adaptive Systems,” in *Proceedings of 11th International Workshop on Requirements Engineering: Foundation for Software Quality*, 2005.

[3] M. Fahrmair, W. Sitou, and B. Spanfelner, “An engineering approach to adaptation and calibration,” in *Modeling and Retrieval of Context MRC 2005*, ser. Springer LNCS, no. 3946, 2006.

[4] W. Sitou and B. Spanfelner, “Towards requirements engineering for context adaptive systems.” in *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, vol. 2. Beijing, China: IEEE Computer Society, 2007, pp. 593–600.

[5] A. K. Dey, “Providing architectural support for building context-aware applications,” Ph.D. dissertation, College of Computing, Georgia Institute of Technology, 2000, director-Gregory D. Abowd.

[6] H. Lieberman and T. Selker, “Out of context: computer systems that adapt to, and learn from, context,” *IBM Systems Journal*, vol. 39, no. 3-4, pp. 617–632, 2000.

[7] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, “Recovery oriented computing (roc): Motivation, definition, techniques,,” Berkeley, CA, USA, Tech. Rep., 2002.

[8] M. Mikic-Rakic, N. Mehta, and N. Medvidovic, “Architectural style requirements for self-healing systems,” in *WOSS’02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA: ACM Press, 2002, pp. 49–54. [Online]. Available: http://portal.acm.org/citation.cfm?id=582138

[9] M. Fahrmair, “Kalibrierbare kontextadaption für ubiquitous computing,” Ph.D. dissertation, Fakultät für Informatik, TU-München, 2005.

[10] M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia, and S. Hariri, “AutoMate: Enabling Autonomic Applications on the Grid,” in *Autonomic Computing Workshop*. IEEE Computer Society, 2003, pp. 48–57.

[11] M. M. Gorlick and R. R. Razouk, “Using weaves for software construction and analysis,” in *ICSE ’91: Proceedings of the 13th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1991, pp. 23–34.

[12] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. W. Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow, “A component- and message-based architectural style for GUI software,” *Software Engineering*, vol. 22, no. 6, pp. 390–406, 1996.

[13] N. Medvidovic and M. Mikic-Rakic, “Architectural support for programming-in-the-many,” TR USC-CSE-2001-506, Tech. Rep., 2001.

[14] E. Mohyeldin, M. Fahrmair, W. Sitou, and B. Spanfelner, “A Generic Framework for Context Aware and Adaptation Behaviour of Reconfigurable Systems,” in *Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC’05)*, 2005.

[15] E. Mohyeldin, M. Dillinger, M. Fahrmair, W. Sitou, and P. Dornbusch, “A Generic Framework for Negotiations and Trading in Context Aware Radio,” in *Software Defined Radio Technical Conference, Phoenix Arizona USA, November*, 2004.