A Roadmap towards Autonomic Service-oriented Architectures

Lei Liu Hartmut Schmeck

University of Karlsruhe, Institute of Applied Informatics and Formal Description Methods (AIFB)
D-76133 Karlsruhe, Germany
Email: {liu | schmeck}@aifb.uni-karlsruhe.de
http://www.aifb.uni-karlsruhe.de/english

Abstract: The advent and proliferation of Service-oriented Architectures (SOA) drives computing infrastructures into a highly interconnected, heterogeneous, and dynamic world. Conventional management tools fail in the attempt to deal with the heterogeneity and the dynamics associated with this type of information infrastructures. More and more researchers try to cope with the complexity, heterogeneity, and uncertainty by using technologies inspired by biological systems. A promising approach for managing such large-scale IT infrastructure is to provide capabilities for self-organization, which – to some extent – is analogous to the human autonomic system (as postulated in IBM's Autonomic Computing Initiative and as extended in the German Organic Computing Initiative). This paper outlines a common view on *Autonomic Service-oriented Architectures* and proposes a way to get such an autonomic infrastructure. An outline of the differences between autonomic service-oriented architectures and other systems with autonomic properties is followed by a discussion of the existing enabling technologies and of missing pieces on the roadmap to a self-organizing infrastructure.

Keywords: autonomic computing, service-oriented architecture, self-organization, organic computing, Web services

1. Introduction

The possibility to expose business capabilities as web services in a platform-independent manner gives organizations the potential to share their capabilities across organization as well as platform boundaries. It facilitates the realization of business processes involving several organizations and, however, turns the Internet into a tightly interwoven network with an increasingly unmanageable technical infrastructure. Together with the underlying technologies, such as hosting environments, hardware, and network components, the complexity of the IT landscape rises into a new dimension, which will barely be manageable with conventionally available tools.

In the context of systems based on service-oriented architectures (SOA), the restricted capabilities of conventional management tools lead to unreliable systems and high costs typically associated with failure and fault recovery. As pointed out by Ganek and Corbi [9], 40% of computer system outages are caused by operator error because of the complexity of today's computer systems that are difficult to be understood. However, the aforementioned issues constitute only part of the complexity related to SOA-based systems and to the business systems built upon them. The problem domains can be categorized as follows:

- *System development*: developing SOA-based systems comprises designing, coding, testing, and deploying the individual parts of the systems. Compared to traditional component-based software engineering, development of software for SOA-based systems is more complex because of the distributed nature of such systems as well as the requirements for adaptability and flexibility of the systems later at runtime.
- *System management*: managing systems embraces tasks such as deploying, configuring, problem solving, resource maintenance, security management, and many other activities for providing services reliably and effectively. Barrett et al. [2] have performed several field studies of the current administrator's work practices and confirmed that system administration is rapidly becoming more difficult as the system complex grows. Furthermore, the loosely-coupled nature of SOA-based systems makes the administration tasks at runtime even more complex.
- *Process management*: the complexity of process management arises from the artifacts involved in business processes: technical systems, people, resources, and so on. The gap between business processes and the underlying SOA-based infrastructure with respect to the different operational goals makes process management even more complicated, since a general understanding of both layers is required for successfully operating the business processes.

Driven by the necessity of reliable systems for daily business, more and more industry vendors are looking for new designs of management systems to cope with the growing complexity in SOA-based systems. Decentralized management approaches, such as the manager-to-manager interface introduced by Liu et al. [15], aim to combine the capabilities of management applications involved in the system landscape to provide federated management. But this is not a sufficient way to construct a generally applicable solution for managing SOA-based systems as long as administrators are still strongly involved in managing the complex, heterogeneous, and dynamic infrastructure. The concept for solving this problem is to give the systems the capability to manage themselves. To find appropriate rules for self-management it looks promising to investigate self-

organizing systems observable in nature. Evolutionary algorithms [7] and ant colony optimization [6] are prominent success stories of bio-inspired methodologies in computer science.

An interesting source for getting the necessary inspiration is the human *autonomic* nervous system: It carries out a large variety of functions, such as temperature and heartbeat regulation, across a wide range of external conditions without any conscious intervention of the human itself. Inspired by this system, in early 2001 IBM introduced the *Autonomic Computing* initiative to enable the creation of an IT computing infrastructure that automates its management in a similar way as the autonomic nervous system. The major characteristic of an autonomic system is the presence of the so-called self-x properties: self-configuring, self-optimizing, self-protecting, and self-healing [9]. In comparison to other similar initiatives, such as the German Organic Computing Initiative, autonomic computing aims at deriving universal concepts for designing, developing, deploying, and managing hardware and software components of large-scale enterprise server systems with a particular focus on dynamically changing environmental and system requirements (see Section 3.1).

This paper identifies specific requirements for turning service-oriented architectures into autonomic systems and lists key software components required to create a self-managing computing infrastructure based on SOA. The paper is organized as follows. Section 2 and Section 3 review the concepts of service-oriented architecture and autonomic computing separately. Section 4 presents the characteristics of SOA-based infrastructures and how autonomic computing might help to cope with their specific requirements. Section 5 addresses the missing software components for realizing self-organization and outlines the roadmap towards an autonomic service-oriented architecture both from a fundamental and an engineering perspective. Section 6 adds some concluding remarks.

2. Service-oriented Architectures (SOA)

The reference model for SOA from OASIS [16] defines a service-oriented architecture as a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. An SOA provides the necessary capabilities to integrate, publish, discover, and manage services. Derived from the concept of object-oriented distributed systems, an SOA follows the design principles for encapsulation, abstraction, and reusability. However, it differs also strongly from the traditional distributed system paradigms, particularly from the perspective of platform-independence. While most object-oriented distributed systems are proprietary and based on some vendor-dependent technologies and platforms, an SOA is completely platform- and technology-independent, thanks to the adoption of widely accepted standards, such as XML, WSDL, etc.

There are various versions of SOA definitions based on different viewpoints in an enterprise computing infrastructure. However, the central concept of all these versions is *service*, which means that the needs of a service consumer are matched with the capabilities brought by the service provider. A service in an SOA can be any possible function that exposes its capability using a prescribed interface in compliance with the SOA standards and acts consistently with the constraints and policies defined in the description. Therefore, the actual implementation of the respective service is not the concern of SOA. This "black box" approach allows changing the implementation details with minimal impact on the service consumer. Furthermore, it allows the integration of legacy applications that are either not network-enabled or not standard compatible for SOA. A service is typically accessible via a network. Among all the services that are currently adopted, web services form the major part.

In an SOA-based computing infrastructure, services are loosely coupled to each other. The only binding between a service provider and a service consumer is a formal service contract. The service contract defines the terms that both the consumer and the provider should follow. For the service provider the service contract basically is a service description that provides the published details about the service, such as service interface, access information, usage definition, etc. For the service consumer, the service has to restrict itself to the terms defined in the service contract, so that the interaction between it and the provider remains unobstructed. This formal binding over the service contract forms the loosely coupled relationship between the provider and the consumer. With *service* as the central concept, an SOA-based computing infrastructure can be modeled as depicted in Fig. 1. The *Technology* layer contains all the enabling technologies and the platforms for Web services. The *Application* layer runs on top of the technology layer and provides the capabilities that can be encapsulated as Web service in the *Service* layer. The processes in the *Process* layer orchestrate the services from the *Service* layer to build composite functionalities out of the ones provided by the services. The processes are consumed either by another process to build composite processes or by a workflow-enabled application.

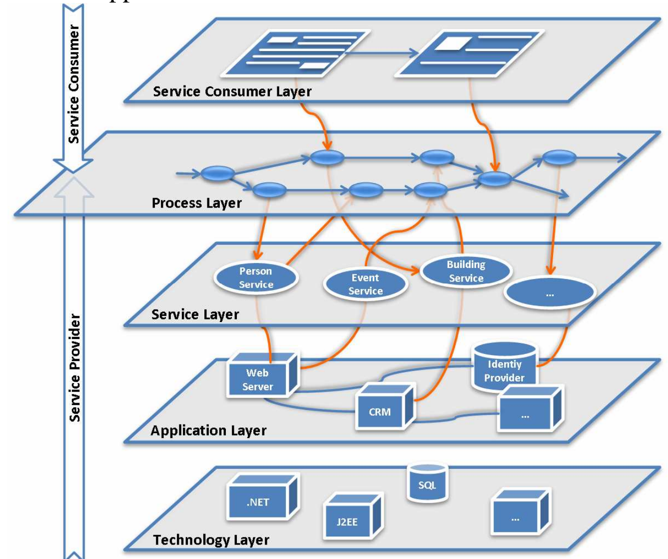

Fig. 1. Abstract model of SOA-based enterprise computing

The SOA-based computing infrastructure is a completely XML-driven architecture. The Web service framework defined by W3C [4] is based on three core specifications: WSDL for service description, UDDI for service discovery and SOAP for message transmission. This basic Web service architecture establishes the foundation for creating loosely coupled Web services that encapsulate isolated business

functionality. Based on these core specifications, service-oriented applications can be built that are within or beyond the boundaries of organizations. But these specifications are not sufficient for building applications in the real world, because they do not address most of the problem domains that distributed systems have to face, such as reliable messaging, security, context, and transactions based on the stateless connections between services. The goal to empower the service-oriented architecture to meet the real world's requirements drives the Web services community to extend the capabilities of the Web services architecture based on the W3C Web service framework. To avoid the extreme cost for developing an entire protocol for each vertical domain the Web service protocol stack is designed as a family of composite protocols. Each protocol defines a fine-grained unit of functionality and can be flexibly reused and combined on demand. In the following, some of the major emerging Web services specifications are listed:

- *Messaging*: a major challenge for distributed computing is reliable communication between messaging partners [5]. To make Web services capable of enterprise level applications, BEA, IBM, Microsoft, and TIBCO have jointly published the *WS-ReliableMessaging* specification to allow messages to be delivered between distributed applications even in presence of software, system, or network failures.
- *Transaction*: the initial set of Web services specifications lacks support for maintaining context across several loosely coupled Web services because the Web services are stateless and work independently of each other. To enable distributed transactions across several Web services, further Web services specifications are proposed, such as *WS-BusinessActivity*, *WS-AtomicTransaction* and *WS-Coordination* that are currently hosted by OASIS Web Services Transaction TC [21].
- *Security*: service-oriented enterprise applications depend on a well-secured communication framework. Diverse specifications have been proposed by industry and standardization organizations. The foundations for the Web service security framework are *XML Signature* [31], *XML Encryption* [32] from W3C and *WS-Security* from OASIS [20]. They establish the security measures along the message transport path and protect the SOAP messages from unauthorized actions.

In recent years, more and more organizations have recognized the value of SOA-based solutions for building agile and flexible enterprise computing infrastructures, not only in the industry but also in the academic field. The work described in this paper is conducted in the context of the *Karlsruhe Integrated Information Management* (KIM [13]) project, which aims at supporting and integrating administrative and educational processes in the university context by adequate IT services. Without abdicating the existing legacy systems in the various faculties and in central facilities, the KIM project provides an SOA-based solution to optimize the collaboration spanning several organizations within the university. To achieve this goal, the project focuses on analyzing and categorizing business processes and their underlying services in the university with respect to various technical and organizational criteria. Based on this analysis, services are consolidated and – through the adoption of Web services standards – a homogenous service layer is accomplished on top of the currently rather heterogeneous technology landscape in the university.

3. Autonomic Computing

The term "Autonomic Computing" has been defined by IBM's Autonomic Computing initiative, which has the primary goal to develop computer systems (in particular, large-scale enterprise server systems) that manage themselves while hiding the increasing system complexity from the end users and even from system administrators. During a keynote presentation at the AGENDA 2001, Paul Horn has compared the concept of Autonomic Computing to the human's autonomic nervous system that regulates the complex human body without self-conscious actions of the human [12]. The vision of Autonomic Computing is to apply the same ability of self-regulation to computer systems, so that one day computer systems can reach the same level of self-regulation as the human's autonomic nervous system.

In the meantime, the concept of Autonomic Computing has evolved from a proposal in Horn's keynote to a widely accepted concept for dealing with the increasing system complexity. As a result, research in industry and academia has focused on various solutions and technologies that exhibit aspects of self-management. However, there is still a lack of a commonly accepted definition of "Autonomic Computing". Paul Lin et al. have tried to establish a common definition for the Autonomic Computing [14]. They carried out a survey on the current publications in the field and identified various definitions for Autonomic Computing. The most commonly referenced definitions contain the following properties that an Autonomic Computing system must have:

- *Self-configuring*: self-configuring is a system's capability to adjust itself dynamically to achieve the desired operational goal, such as performance, reaction time, etc. The self-configuration may assist in self-healing, self-optimizing and self-protecting by dynamically responding to changes in the environment.
- *Self-healing*: from the perspective of reactive systems, self-healing is the capability to discover, diagnose, repair, and recover from system faults when they occur. From the perspective of predictive systems, self-healing contains mechanisms to predict and thereby prevent system faults from happening by monitoring the vital parameters of the system.
- *Self-optimizing*: self-optimizing refers to the capability to measure the current system performance against the predefined objectives and to attempt to improve the performance by efficiently controlling the allocation and utilization of resources.
- *Self-protecting*: self-protecting describes the capability of a system to anticipate and detect external malicious attacks and to protect itself in case of attacks. It means that the system must be aware of potential threats and be able to take actions to avoid completely or at least mitigate partly the affects caused by the external attacks.

To support the functional properties listed above, an Autonomic Computing system should be aware of itself (self-awareness) and of the environment around it (context-awareness). The system should monitor its internal state by collecting management information from its functional components and evaluate the collected data to identify its

vital status. Furthermore, a network-enabled system is not isolated from its environment. For instance, a Web service is related to its hosting environment, to other Web services in the business process that call it or are called by it. More or less, the functional state of all the related systems has impact on the system itself. Therefore, an Autonomic Computing system should know the way to interact with neighboring systems for sharing functional state information. To achieve cooperation between different systems in a possibly heterogeneous environment, the Autonomic Computing system must implement open standards to enable an unobstructed communication with other systems. In some publications in the field, this capability is referred to as "openness" of a system.

A major architectural aspect of an Autonomic Computing system consists of control loop functionalities that contain the following four steps: *monitor*, *analyze*, *plan,* and *execute (MAPE)*. By *sensors* that connect with the managed components, the *monitor* function collects data, for instance, metrics, from the managed components, and filters the data, aggregates it and reports the details to the *analyze* function. The *analyze* function correlates the data being reported and tries to model complex situations from such data. The result of the analysis is consumed by the *plan* function, which selects or constructs actions based on the analysis and the predefined operation policies. The *execute* function controls the execution of the action plan using *effectors*, which are connected to the managed components.

3.1 Related Work

The idea to simplify the management of technical systems by applying nature-inspired mechanisms has led to a set of industrial and academic projects, as Mazeiar and Ladan have reviewed in their publication [17]. In the following, we review the concepts of two other initiatives that are missing in the work of Mazeiar and Ladan.

3.1.1 Organic Computing

Similar to the concept of Autonomic Computing, Organic Computing (OC) is an emerging paradigm for coping with the increasing presence of large collections of intelligent objects in various areas of our daily life, capable to communicate and to interact. In particular, Organic Computing outlines the vision of technical systems that exhibit various self-x properties like self-configuration, self-optimization, self-healing, self-explanation, and self-protection, capable to learn about their environment over time, survive attacks and breakdowns, adapt to their users, and react sensibly even if they encounter a new situation for which they have not been programmed explicitly. OC systems should be designed with respect to human needs, they have to be trustworthy and robust, adaptive, and flexible. [26]. Because of their life-like properties, these systems are called *organic systems.* Organic Computing emphasizes that future technical systems will inevitably have the capability to self-organize. Therefore, one has to address the major challenge to guarantee that, nevertheless, these systems will always adhere to externally given objectives and constraints while adapting to dynamically occurring changes in their environment. In the German priority research program on Organic Computing [34], the major focus is on

- fundamental investigations of the effects of emergence on the controllability of self-organizing systems,
- investigating systems occurring in nature which exhibit forms of self-organization in order to identify behavioral patterns that might be transferable into technical systems,
- the development of generic architectures for realizing organic systems,
- the design and investigation of specific technical applications as prototypical organic systems.

Obviously, OC systems satisfy the requirements of Autonomic Computing.

3.1.2 Organic IT

Organic IT has been proposed by Forrester Research in 2002 [10] to increase the IT efficiency and maximize the business value of enterprise computing infrastructures. The vision of Organic IT is to build computing infrastructures on redundant components that automatically share and manage enterprise-computing resources, such as software, processor, storage, etc, across all applications within a datacenter. To deal with the heterogeneity in the computing infrastructure, the key concept of Organic IT is abstraction, the way to hide complexity behind a simple interface and to combine such simple interfaces into an improved whole. Derived from this principle, the computing infrastructure synthesizes the four key layers: network, storage, processors and software. All the four layers are managed by a single management console on an exception-driven basis.

4. Enabling Autonomic Computing in SOA

As stated in the introduction, the complexity of an SOA-based system derives from several fields, from software development at design time to system management at runtime, and grows rapidly due to the increasing number of services and the heterogeneous environments that the services rely upon. Applying the concepts of Autonomic Computing to the systems is promising for coping with the complexity while reducing the management overhead for administrators. Above all, Autonomic Computing can support and enable service management and service integration in concert with the principles of SOA. However, the concept of Autonomic Computing is too coarse to fit the requirements that an SOA-based system may have. In this section, we discuss the functional requirements of an SOA-based system on Autonomic Computing and show, how an autonomous service-oriented architecture might look like.

In the context of service-oriented architectures, the computing infrastructure shows some level of stability due to the service level agreements (SLA) established between service providers and service consumers. Furthermore, the model applied in a business process commits a service provider to its service consumer(s) and establishes a relationship between them. From this point-of-view, an autonomic service-oriented architecture operates in a comparatively stable and closed environment and emphasizes the automated and robust management of the architecture rather than the dynamic (re-)organization of business processes.

4.1 Functional Requirements

In the abstract SOA model in Fig. 1 five abstraction layers have been identified: technology, application, service, process, and service consumer. Across all the layers, services are the central components in the architecture, where units of business capabilities are encapsulated. The process layer above the service layer and the application layer directly beneath are the layers, which have direct functional relationships with the service layer. Due to the central role of services, we discuss the functional requirements of SOA based on a service and its relationship to other components in the architecture. As the basic element in the architecture, which should be managed, a *service* constitutes an autonomic element [33] that is responsible for its own behavior and cooperates with other autonomic elements in accordance with the global operational goal.

In an exemplary way, Fig. 2 depicts an SOA with services as the central elements. Vertically, a service is based on applications in the Application Layer, for instance, a Web server is the hosting environment for HTTP-based web services. Processes in the Process Layer model the relationship between services in the Service Layer and call the services to invoke the corresponding business functionalities. Horizontally, a service has a cooperation relationship with other services, to which it provides service or which provide services to it.

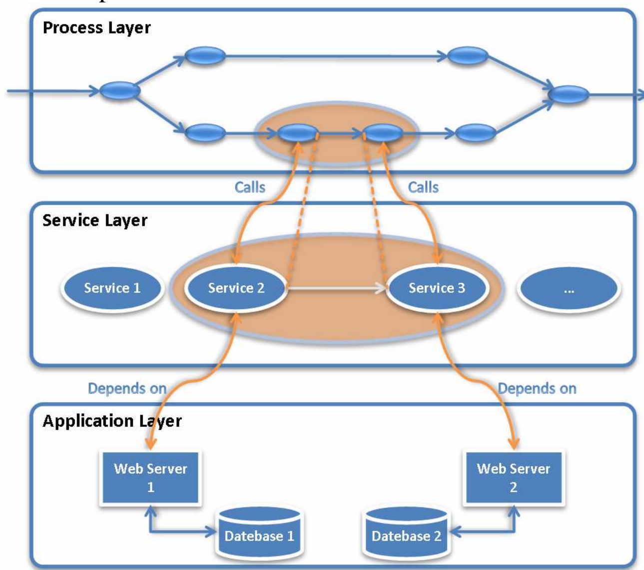

Fig. 2. Service as the central element in SOA

To enable the interaction with other elements in a networked environment, it is required that *a service needs to know itself* (R1). The knowledge of a service about itself can exist at several levels. At the meta-level, a service should know its functionality, its interface to the external world, etc. and a way to describe itself. The Web Service Description Language (WSDL) standard is an example for such a meta-model, which can be used to describe the interface of a service. A further example of such a meta-model is OWL-S [30], which is an Ontology Web Language (OWL)-based web service ontology and describes what a service does, how it works, and how to access the service. At the instance level, a service should have detailed knowledge about its components, its status, and its dependencies with other elements in the SOA. This is the basic requirement for a managed service in the SOA to be self-aware and context-aware.

Next, it is required that *the service should be able to expose meta-level information and at least part of instance-level information to other elements in the SOA on request* (R2). The meta-level information is crucial for other elements to determine the capabilities and access information of the current service, in case that a relationship should be built between them. The exchange of instance-level information, such as performance, metrics, logs, is another key requirement to keep a service context-aware. From the information exchanged between a service and its dependencies, a service can get an overview about its environment and take actions, if necessary, to ensure its operational goal. The exchange of instance-level information can take place voluntarily and in combination with a distributed reputation system, like Obreiter et al. have proposed in their work for a P2P environment [24]. If the instance-level information is critical for sharing with other elements in the architecture, the exchange of such information can take place upon agreements that are established as the two elements enter into a new relationship.

It is required that *the service should be able to establish and maintain relationships with other elements in the SOA* (R3). The meta-level information discussed in requirements R1 and R2 is the basis for the new relationship with other autonomic elements in the SOA. There are mainly two types of relationship in the SOA. The first one is the “provider/consumer” relationship, which is regulated by service level agreements between the service consumer and service provider. In this case, both sides must understand the terms in the agreements and, if necessary, negotiate the terms in the agreements with each other. Once two parts enter into an agreement, they must abide by the terms defined in the agreement to maintain their relationship. The second type of relationship in SOA is the dependency relationship, for instance, the “depends on” or “calls” relationships as illustrated in Fig. 2.

It is required that *the service should be context-aware* (R4). A service should at least know its direct neighborhood in the SOA. In other words, a service must know about all its neighbors with “provider/consumer” relationships. Furthermore, it has to know all the components in the SOA, which it depends on or which depend on it. Through the regular exchange of information with its direct neighborhoods, the service can discover its environment and take necessary actions, if the environment changes. Furthermore, a service must have knowledge about the infrastructure services available in the SOA, which it may make demands on, if necessary, for instance, a registry infrastructure service helping the services in an SOA to find one another.

It is required that *the service should be able to control its own behavior to meet its own operational goal* (R5). The service has an operational goal that can be specified initially at starting time or be specified by a related element in the architecture as part of an agreement. The agreements between the service and other elements to establish the relationships are part of the operational goal to meet. Furthermore, if the service is involved in a business process, there will be some global goals for the whole business process. In this case, the service has to adjust itself to meet the global goal being given. Generally, there are two possible ways to adjust a

service's behavior: Either it can configure its own parameters or it can rely on its dependencies in the SOA. For example, a web service can adjust its performance by controlling the appropriate parameters on its hosting environment, such as the caching time on the web server.

Furthermore, it is required that *the service should be able to take an external directive and execute it, if applicable* (R6). For a service in an SOA, external directives are only requests and the requestor cannot assume that the service will execute the directive. Depending on the policies and the type of the directive, a service can decide how to deal with the directive. If it is an administrative directive from an element in the SOA that has sufficient authority to issue directives, the service will take the directive as a command. If a service receives conflicting directives from different elements in an SOA, it can try to resolve the conflict by itself, or it can refer to other elements in the SOA for help [33].

4.2 Required Interfaces

To achieve the functional requirements stated above, open standards must be adopted to enable the interactions between the elements in the SOA. As stated before, an SOA-based system may have a heterogeneous computing infrastructure spanning several organizations. A proprietary implementation of autonomic elements can only deal with part of the SOA, which is compatible with the implementation. Therefore, all the implementations should be based upon open standards, from the web service standards to management standards, for interoperating in a heterogeneous system environment.

Web services define a set of interfaces and specifications to achieve various functionalities based on the basic W3C Web service framework [4]. To realize the functional requirements discussed in this section, the autonomic elements in the SOA need additional interfaces as well:

- *Metadata interface*: this interface allows a service to expose its meta-level information to other elements in an SOA that intend to establish a relationship with the service. The information exposed by this interface can also be used e.g. by the service registry to find services with certain criteria.
- *Performance interface*: this interface allows a service to expose its instance-level information to other elements in an SOA at runtime. As this service may expose critical information, which is not viewable by any service in the SOA, the autonomic elements that call this interface must have either the appropriate administrative relationship with the service or an agreement with the service about sharing the information.
- *Binding interface*: this interface allows a service to establish a "provider/consumer" relationship with other services. Through the binding interface, a service can negotiate the terms in the service level agreement with the service requestor. The service requestor receives either a confirmation or a rejection, if they cannot come to an agreement about the service level parameters.
- *Administrative interface*: this interface allows a service to receive administrative directives from other autonomic elements in an SOA at runtime. The service requestor authenticates itself at the service to show that it has sufficient authority to issue directives. The service can make its own decision about what to do with the directives being received. The directive can be interpreted individually as command or suggestion, depending on the relationship between the service and the requestor.

4.3 The Self-x Properties in Autonomic SOA

As aforementioned, currently, the self-x properties (self-configuring, self-healing, self-optimizing, and self-protecting) are used to evaluate systems with respect to autonomic computing. In this section, we discuss how the functional requirements for SOA are mapped to the self-x properties.

Table 1. Self-x properties in autonomic SOA
(* indicates that there is a correlation between two terms)

	R1	R2	R3	R4	R5	R6
Self-configuring	*	*	*	*	*	
Self-healing	*				*	
Self-optimizing	*	*	*	*	*	*
Self-protecting	*				*	

Table 1 shows an overview of the correlations between the functional requirements and the self-x properties.

- *Self-configuring*: to achieve self-configuration, a service needs to be self-ware and context-aware. The services share information about the environment in the SOA through the exposing interfaces. To adapt to the changes in the environment, the service has to control its behavior depending upon its operational goal and the new environmental conditions.
- *Self-healing*: self-healing requires that the service be self-aware. Based on the instance-level information that it collects at runtime, such as performance, metrics, logs, a service can control its behavior to prevent faults from happening or to recover the faults, if they have occurred.
- *Self-optimizing*: to achieve self-optimization, a service needs to be aware of its own state and the state of its environment by analyzing the data collected via the exposing interfaces. Based upon this information, a service can either control its behavior or issue directives to other elements in the SOA, to which it has relationships.
- *Self-protecting*: To protect itself, a service needs to be self-aware and, in case that external malicious attacks are detected, the service has to control its behavior to protect itself. For example, it could reject all the requests from hostile service requestors or it could also go offline temporarily to avoid the attacks.

In this section, we have discussed the functional requirements on an autonomic service-oriented architecture and how these requirements are mapped to the self-x properties of Autonomic Computing. In the following section, we discuss how such an autonomic service-oriented architecture can be designed, developed, and operated.

5. The Way to an Autonomic SOA

The design and development of an autonomic service-oriented architecture is a holistic process that covers research

for software and systems engineering (SE) as well as for artificial intelligence (AI) [29]. The engineering approach concerns itself with mechanisms to engineer autonomic capabilities into the individual systems, while the artificial intelligence approach implies utilization of algorithms and processes to achieve autonomic behaviors of the components of an SOA. In the following subsections, we discuss how to realize an autonomic SOA based on these two approaches and the functional requirements addressed in the last section.

5.1 State-of-the-Art

Several research efforts have contributed to enable self-x behaviors in an SOA-based system. Such approaches aim at either some particular self-x property or at particular layers in an SOA-based system. For instance, Sherif et al. introduced an approach that enables the self-x properties in an SOA-based system by using dedicated autonomic Web services at runtime [27]. Each autonomic Web service implements the MAPE control loop and provides some autonomic functionality, such as self-healing, to other functional Web services. Instead of providing the self-x behaviors to the complete SOA-based system, as proposed in this paper, they focused mainly on the service layer of an SOA-based system. Pautasso et al. have proposed to create a reactive architecture across both the service layer and the process layer by a MAPE control loop [25]. Their system monitors the performance of processes running within an SOA-based system. Once workload variations are detected by the system, it alters its configuration in order to optimally use the available resources of the service layer.

Just like both systems introduced above, most of the approaches in the field focus only on part of the autonomic aspects for an autonomic SOA. From our point-of-view, an autonomic SOA requires a comprehensive approach including all the self-x properties and all the layers of an SOA-based system. For example, it is assumed that a business process in the process layer rests upon a set of Web services that are hosted by an application server in the application layer. In this scenario, any negative workload variation of the application server may cause longer responding time for the Web services, which again may lead to failures in the business process due to the violation of service level agreements between the process and the Web services.

IBM provides the Autonomic Computing toolkit that can be used to build prototypes with autonomic behaviors. The toolkit contains building blocks for enhancing autonomic capabilities including problem detection, common system administration, and system installation and deployment into the prototypes. For translating legacy log entries into the common event format, the toolkit contains an adapter, the Generic Log Adapter (GLA), to include legacy systems into the autonomic architecture without requiring such systems to change the way they create the log files. Based on the events and tracing being collected, the Log and Trace Analyzer (LTA) analyzes and correlates the log entries. In case that an incident is detected, LTA consumes the symptom database and delivers an array of objects representing the solutions and directives. The Autonomic Management Engine (AME) provides a reference implementation of an autonomic manager. At runtime it monitors the system resources, sends aggregated events, and performs corrective actions for problems detected.

5.2 Web Services Standards

One of the challenges for building an autonomic SOA is the heterogeneous computing infrastructure that the SOA relies on. The only way to deal with this heterogeneity is to keep the autonomic SOA open by adopting widely accepted industry standards. In Section 2, we have briefly introduced the concept and the evolution of service-oriented architectures. To empower the Web service architecture to provide more functionalities for building reliable, secure, and trusted service-oriented applications, a set of new Web service specifications has been proposed by various organizations. For building an autonomic SOA, the following specifications are of particular interest:

- *Web service discovery* is the key for automatically connecting to existing services in the infrastructure without intervention of administrators or operators. W3C has identified two possible types of discovery mechanisms for Web service architecture [4]: a registry-based centralized approach and a peer-to-peer based distributed approach. The static discovery applies the centralized approach and provides the necessary capabilities for looking for the potential cooperation partner at a well-known location. The UDDI registries apply this approach and serve as the directory for registering and querying existing Web services. The dynamic discovery applies the peer-to-peer approach. In dynamic discovery, there is no well-known location for querying Web services. To discover an appropriate Web service, the requester broadcasts a request to all available listeners. *WS-Discovery* defines this approach and specifies procedures to announce and discover Web services using multicast messages [3].
- *Metadata exchange*: in general, information about a Web service is collectively referred to as *metadata*. Web services use a lot of metadata, such as WSDL, XML Schema, to describe a particular Web service interface. Web service discovery alone is not sufficient for automatically building relationships between services. A service consumer needs more meta-information about the service provider to enable the bootstrap communication with it. A new Web services specification, *WS-MetadataExchange*, allows a service provider to deliver metadata to its potential consumers via a predefined Web services interface, both at design time as well as at runtime [1].
- *Web Service Management*: the term “management” has two aspects for a SOA-based system: management using a Web service and management of a Web service. Currently, both aspects are considered by OASIS. Each of these aspects has a separate specification: *Management Using Web Services* (MUWS) [23] and *Management of Web Services* (MOWS) [22]. The core component in the specifications is the *Web Services End Point*. It interprets Web services messages and provides access to a backend *manageable resource*. A *manageable resource* can have a number of capabilities, each of which have distinct semantics, and provide these capabilities outwards via the Web Services *End Points.* Another similar proposal in the

field is *Web Services for Management* (WS-Management, former *Web Service for Management Extension* [18]), which is a joint publication of AMD, Microsoft, Sun et al. Comparing to the management specifications of OASIS, WS-Management can be considered as a lightweight version of the other two specifications from OASIS, which makes it suitable for use in small devices with restricted resources.

5.3 Building Autonomic Service-oriented Architectures

The way to an autonomic service-oriented architecture is rather evolutionary than revolutionary, just as proposed by IBM in their Autonomic Computing Initiative [9]. In their vision for autonomic computing, the path to an AC-enabled system can be thought of five levels, starting at *basic* and continuing through *managed*, *predictive*, *adaptive* and finally *autonomic*. The five levels describe the transition from a basic system to a completely autonomic system by two aspects: to enhance a unified system management of the entire infrastructure step-by-step, and to increase the ability to make decision autonomously based on the environment information being collected. The crucial task to obtain an AC-enabled system is to build a managed infrastructure embedding the MAPE control loop with the necessary components as well as interfaces.

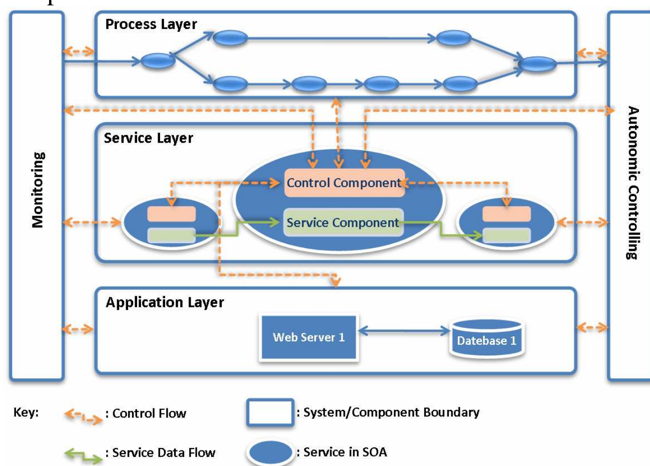

Fig. 3. Coarse architecture of an autonomic SOA

5.3.1 Autonomic service-oriented architecture

Fig.3 coarsely illustrates the architecture of an autonomic SOA-based system. In general, the architecture exhibits its autonomy at two levels: one at the service level providing autonomic capabilities directly to the services, and the other one at the infrastructure level that provides the autonomic capabilities to the whole SOA-based systems. The service-level autonomy is provided by the service itself. With services as the central elements in the autonomic SOA, each service employs two functional parts. One is the "Service Component" in the service, which provides the desired business capabilities as Web services to external consumers. The other is the "Control Component" that interacts with the service's direct neighbors in the infrastructure, on which the service depends functionally. Each control component implements the four functional interfaces that were discussed in Section 4.2. Through these interfaces, the control components in the service layer as well as in other layers can interact with one another, so that a service can dynamically control its behavior based on the information it observed in its direct neighborhood. In this context, the control component gives a service the ability to adapt to local changes.

Due to the fact that the control component has a limited view in terms of environment information, it can only make local decisions, in some cases sub-optimally or even wrongly. In this case, the infrastructure-level autonomy helps the service to improve the decision. The two cross-cutting layers in Fig. 3, "Monitoring" and "Autonomic Controlling", are responsible for the infrastructure-level autonomy. The monitoring-layer communicates with each control component in the infrastructure and thereby gets an updated overview of the entire infrastructure at any time. Based on this global and its global policy, the autonomic controlling layer can control the affected infrastructure components to ensure the global operational policy.

5.3.2 Building the control components

The interfaces discussed in Section 4.2 within the MAPE control loop are required for realizing the autonomic capabilities at the services of an SOA-based system. However, the SOA implies a heterogeneous service landscape that cannot be instrumented with autonomic capabilities in the same way. The elements in an SOA should be instrumented individually according to the possibilities available for integrating external functionalities:

- For existing applications and services in an SOA, it is only possible to apply a decoupled agent to the applications, because the code of these applications is not available for modification. The external agent implements the interfaces for the applications and services in the SOA and handles the autonomic behavior for the applications. In this case, an approach is needed that can incorporate self-managing features into applications without modifying them. The use of Aspect-oriented Programming (AOP) allows treating the autonomic functionalities as a concern, as Hoi et al. have demonstrated in their experiment [11]. The development of such functionalities is separate from each other and the integration of such functionalities into the application is selective, based on the functional requirements that the agents have.
- For new services in an SOA, it is much more efficient to include the interfaces directly in the service itself to achieve a better control of the autonomic behavior of the elements. In other words, the autonomic functionalities are tightly coupled with the services and in this way provide direct control of the service's behavior. The implementation of the interfaces is straightforward and does not differ severely from the implementation of normal services.

The implementation of an autonomous SOA by instrumentation of the elements in the SOA individually may be difficult and complex. New approaches from software engineering using model-driven architecture (MDA) and domain specific language (DSL) [19, 28] may help to reduce the complexity and increase the manageability of the SOA. With MDA, it is possible to build the autonomic functionalities into the services directly during the

development process. This approach complies with the tightly coupled autonomic functionalities in the services and accelerates the development of the autonomous SOA. As prerequisites for the MDA development, models and domain specific language are required for describing the artifacts in the autonomous SOA. With the help of them, an autonomous SOA can be expressed in DSL and the expression in DSL in turn can help to generate action plans about setting up the elements in SOA with autonomic capabilities. The expression in DSL can even be used by a dedicated DSL compiler to generate services for SOA, which have the built-in autonomic capabilities directly at design time.

5.3.3 Controlling the behavior

Without evaluation of the data being collected at runtime, an autonomic element in an SOA can never understand itself and the environment around it. The evaluation of the data can be classified into two categories: the first category contains the analysis of live data, for example the response time of a request or a request error sent back by the provider. The second category contains the analysis of historical data throughout a time span. This evaluation category is crucial for the functionalities of an autonomic service-oriented architecture, especially for self-healing and self-optimizing, because the autonomic elements in the SOA need the result of the evaluation to be self-aware as well as context-aware. The evaluation of the historical data contains event correlation, which interprets multiple events and gives them a collective meaning that represents an event at a higher level. To correlate events, correlation rules are needed that identify which events to correlate. Machine learning, data mining, and other technologies can be used to help discover correlation rules and interpret the correlative events [29].

In the SOA-based computing infrastructure, there is normally a global operational goal for the whole SOA. Such an operational goal can be an operational policy for a business process or for a set of services that are related to one another. Moreover, the operational goals are normally human-centric, in other words, the operational goal might be expressed in natural language and should be converted to machine-readable instructions at first. Another challenge is to split the global operational goal into the local goals for the various autonomic elements in the SOA, which normally takes place at the beginning of service deployment and is not time-critical for the performance of the service. A promising approach for this process is the utilization of evolutionary algorithms to generate action plans from the high-level operational goal for each of the autonomic elements in the system.

6. Conclusion & Outlook

This paper described a roadmap for applying the self-x capabilities as postulated in autonomic computing to SOA-based computing infrastructures. An SOA-based infrastructure exhibits specific functional properties in comparison to other self-organizing systems. Therefore, the concept of autonomic computing has to be adapted to be applicable for an SOA-based system.

Specifically, the complexity present in SOA-based computing infrastructures and the need to provide self-x properties for SOA-based systems have been addressed. After a review of the fundamentals and features of Service-oriented Architectures and Autonomic Computing, the functional requirements of SOA-based systems were outlined with respect to Autonomic Computing and how these requirements are mapped to the self-x properties. Based on the functional requirements the architecture with the necessary system components was introduced coarsely and it was outlined which technologies from the areas of software engineering and artificial intelligence could be utilized to design, develop, and operate autonomic SOA-based systems.

However, there are several issues for autonomic SOA, which have not been addressed in this paper. One of them is the delegation of directives for a managed element in the system. A managed element, for instance, an autonomic Web service, may receive requests from other managed elements, for changing its configuration. In order to make the decision about how to behave in such a situation, the Web service should be supported by an authorization model and a predictive analysis of the possible impact to itself as well as to other dependent system components. This aspect is currently missing. Another issue that has to be investigated is the impact of the business layer on the service layer at runtime. The considerations in this paper are service-centric. In other words, the services are considered as the central elements in the SOA. This simplifies the business processes in the way that they are treated as the composition of several related services in accordance with some rule(s). Indeed, business processes are more than that. Business processes may employ activities other than just (Web) services. How the business process management can be combined with the autonomic elements in the autonomic SOA-based system remains an interesting aspect to investigate in further work.

Currently, we are in the early stage of work for building an autonomic SOA. In the context of the KIM project, several ideas outlined in this paper are explored. For example, a service map is implemented, called *i2map*, for monitoring Web services being deployed in the KIM system landscape [8]. At runtime, i2map monitors the operational status of the services dynamically and provides an overview on the status of the service layer in the KIM infrastructure at any time. Future work is concerned with the further evaluation of the concept of autonomic Service-oriented Architecture by building prototypes that quantifiably demonstrate the functional requirements as well as the self-x properties for autonomic computing. Furthermore, advanced algorithms and methods from AI should be evaluated for their usability with regard to realize the adaptive autonomic elements in SOA.

Finally, it would be necessary to validate the claim of autonomic computing that self-x properties actually lead to predictably more reliable system behavior while significantly reducing the management complexity.

References

[1] K Ballinger, D Box, F Curbera, et al., Web Services Metadata Exchange (WS-MetadataExchange), 2004, http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-metadataexchange.pdf

[2] R Barrett, P P Maglio, E Kandogan, et al., Usable autonomic computing systems: the administrator's perspective, International Conference on Autonomic Computing, New York, NY, USA, 2004, pp. 18-25.

[3] J Beatty, G Kakivaya, et al., WS-Discovery, 2005, http://msdn.microsoft.com/library/en-us/dnglobspec /html/WS-Discovery.pdf
[4] D Booth, H Haas, F McCabe, et al., Web Services Architecture, 2004, http://www.w3.org/TR/ws-arch/
[5] S Cohen, A Geller, C Kaler, et al., Reliable Messaging in SOA, 2004, http://msdn.microsoft.com/webservices/ webservices/understanding/specs/default.aspx?pull=/library/en-us/dnglobspec/html/ws-rm-soa.asp
[6] M Dorigo and T Stützle, Ant Colony Optimization, MIT Press, 2004.
[7] A E Eiben and J E Smith, Introduction to Evolutionary Computing, Springer, 2003.
[8] M Gaedke, J Meinecke and M Nussbaumer, i2Map - An Approach to Model the Landscape of Federated Systems IEEE International Conference on Web Services (ICWS'05), Orlando, Florida, USA, 2005, pp. 797-798.
[9] A G Ganek and T A Corbi, The dawning of the autonomic computing era, IBM SYSTEMS JOURNAL, Vol. 42, No. 1, 2003, pp. 5-18.
[10] F E Gillett, C Rutstein, G Schreck, et al., Forrester Research Report: Organic IT, 2002, http://www.forrester .com/Research/PDF/0,5110,14136,00.pdf
[11] C Hoi and C C Trieu, An approach to monitor application states for self-managing (autonomic) systems, 18th annual ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications, Anaheim, CA, USA, 2003, ACM Press.
[12] P Horn, Autonomic Computing: IBM's Perspective on the State of Information Technology, 2001, http://www-03.ibm.com/industries/government/doc/content/bin/auto.pdf
[13] KIM, Karlsruher Integrated InformationsManagement, 2006, http://www.kim.uni-karlsruhe.de
[14] P Lin, A MacArthur and J Leaney, Defining autonomic computing: a software engineering perspective, Australian Software Engineering Conference (ASWEC'05), Brisbane, Australia, 2005, pp. 88-97.
[15] L Liu, M Gaedke and A Koeppel, M2M interface: a Web services-based framework for federated enterprise management, IEEE International Conference on Web Services, Orlando, Florida, USA, 2005, pp. 774-782.
[16] C M MacKenzie, K Laskey, F McCabe, et al., Reference Model for Service Oriented Architecture 1.0, Public Review Draft, 2006, http://www.oasis-open.org/ committees/download.php/16628/wd-soa-rm-pr1.pdf
[17] S Mazeiar and T Ladan, Autonomic computing: emerging trends and open problems, Workshop on Design and evolution of autonomic application software 2005, St. Louis, Missouri, 2005, ACM Press, pp. 1 - 7.
[18] Microsoft, et al., Web Services for Management (WS-Management), 2005, http://msdn.microsoft.com /webservices/understanding/specs/default.aspx?pull=/library/en-us/dnglobspec/html/wsmgmtspecindex.asp
[19] M Nussbaumer, P Freudenstein and M Gaedke, The Impact of DSLs for Assembling Web Applications, to appear in Engineering Letters, Special Issue on Web Engineering, Intl. Association of Engineers, 2006.
[20] OASIS, OASIS Web Services Security (WSS) TC, http://www.oasis-open.org/committees/tc_home.php ?wg_abbrev=wss
[21] OASIS, OASIS Web Services Transaction (WS-TX) TC, http://www.oasis-open.org/committees/tc_home.php ?wg_abbrev=ws-tx
[22] OASIS, Web Services Distributed Management: Management of Web Services (WSDM-MOWS) 1.0, http://docs.oasis-open.org/wsdm/2004/12/mows/cd-wsdm-mows-1.0.pdf
[23] OASIS, Web Services Distributed Management: Management Using Web Services (MUWS 1.0) Part 1 - Architectural Concepts and Required Components, http://docs.oasis-open.org/wsdm/2004/12/muws/cd-wsdm-muws-part1-1.0.pdf
[24] P Obreiter and B Koenig-Ries, A New View on Normativeness in Distributed Reputation Systems Beyond Behavioral Beliefs, to appear in Proceedings of the Forth International Workshop on Agents and Peer-to-Peer Computing, Utrecht, Netherlands, 2005.
[25] C Pautasso, T Heinis and G Alonso, Autonomic execution of Web service compositions, IEEE Intl. Conf. on Web Services, Orlando, USA, 2005, pp. 435-442.
[26] H Schmeck, Organic computing - a new vision for distributed embedded systems, 8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Seattle, WA, USA, 2005, pp. 201-203.
[27] A G Sherif and Z Amir, Towards autonomic web services: achieving self-healing using web services, ACM SIGSOFT Software Engineering Notes, Vol. 30, No. 4, 2005, pp. 1-5.
[28] S Shetty, S Nordstrom, S Ahuja, et al., Systems integration of large scale autonomic systems using multiple domain specific modeling languages, 12th IEEE Intl. Conf. on the Engineering of Computer-Based Systems, Greenbelt, Maryland, 2005, pp. 481-489.
[29] R Sterritt, State of the Art: Autonomic computing, Innovations in Systems and Software Engineering, Vol. 1, No. 1, 2005, pp. 79-88.
[30] W3C, DAML, http://www.daml.org/services/owl-s/
[31] W3C, XML-Signature Syntax and Processing, http:// www.w3.org/TR/xmldsig-core/
[32] W3C, XML Encryption Syntax and Processing, http:// www.w3.org/TR/xmlenc-core/
[33] S R White, J E Hanson, I Whalley, et al., An architectural approach to autonomic computing, Intl. Conf. on Autonomic Computing, New York, USA, 2004, pp. 2-9.
[34] DFG priority research program "Organic Computing", http://www.organic–computing.de/SPP

Author Bio

Lei Liu is currently a Ph.D. student at the Institute AIFB of the University of Karlsruhe. His research interest focuses on systematic engineering of self-organizing systems with an emphasis in service-oriented architecture and Autonomic Computing.

Hartmut Schmeck holds a Chair of Applied Informatics at the Institute AIFB of the University of Karlsruhe. His major research areas are bio-inspired methods in optimization, parallel and distributed algorithms, and self-organization in complex systems. He is the coordinator of the German priority research program on Organic Computing.