Architecture Analysis of Service Oriented Architecture

Benjamin Kanagwa
Faculty of Computing and
Information Technology
Makerere University
P. O. Box 7062, Kampala, Uganda
Email: bkanagwa@cit.mak.ac.ug

Ezra K Mugisa
Faculty of Computing and
Information Technology
Makerere University
P. O. Box 7062, Kampala, Uganda
Email: emugisa@cit.mak.ac.ug

***Abstract*— Service Oriented Architecture (SOA) enables software systems to possess desirable architecture attributes such as loose coupling, platform independence, inter-operability, reusability, agility and so on. It has been widely published that SOA and services have their origin in other advances in software architecture. However, they do not show the exact features that are inherited and what makes SOA a unique approach. In this paper, we take an architecture approach and analyse SOA in relation to other advances in software architecture and outline the features inherited from its predecessor. The paper also relates the concerns of SOA and the enhancements made on the features from the predecessors in order to satisfy its concerns and make SOA an attractive approach. In this way, we facilitate the understanding of SOA in terms of existing architecture concepts that are already well understood. We believe that a good understanding of SOAs in terms of other advances in software architectures will help to reap its enormous benefits.**

keywords: *service oriented architecture, software architecture and components*

I. Introduction

SOA has gained more popularity of recent, largely due to web services and partially due to natural evolution in software engineering [1]. At the same time, there is a systematic progression from object oriented systems to distributed objects, components and then service oriented systems. This evolution in software engineering shows a systematic move from tightly coupled and platform specific systems to more robust loosely coupled platform-independent systems.

Despite this progression, and widespread interest in SOAs, most publications on the subject fail to explain SOA or simply assume it merely defines a synonym for a stack of extensible markup language (XML) web service protocols and standards [2]. Stal [2] argues that developers need to understand the service oriented paradigm from the architecture perspective in order to leverage SOA implementations. This lack of understanding has already manifested itself in faulty implementation of SOA [3]. We argue that these faulty implementations that carry remote procedure call sentiments can only be addressed by understanding the core principles of SOA. Since we can not divorce services from SOA, we start by addressing the problem from the core. The focus is to provide an understanding of SOA in terms of well established concepts in software architectures.

A strong relationship between existing architecture approaches and SOA has been reported in the literature. For instance [4] argue that: "both component-based and service oriented development, use the same technologies and same or similar principles in architecting of software systems". We would like to add that SOA builds on existing knowledge from other architecture approaches. Our interest is to answer the question; which old system does SOA compare with or extend? Or what does a service extend? To put the question in another way, which architecture features does it build on? This question has been answered by [5] and [6]. In [5] a comparison between Web services and CORBA is made while [6]; suggests that some elements of component technology apply to services. What [6] and [5] do not answer however, is what exactly do we carry forward to SOA? We therefore study the relation between SOA and other advances in software architectures. We show which features are inherited as is and those inherited with relaxations. We show how such features contribute to the resulting SOAs.

Given that the software architecture discipline has had significant advancement, it is important to establish how SOA has built on these advances.

A. Service and SOA definitions

At this point we provide working definitions for *SOA* and *service.* We prefer to define a system (application) that is considered to have a SOA. From this definition, we can establish the principles built within SOA systems. A service is defined as an independent set of functionality that exposes a resource for use by others and is invoked directly under the conditions it specifies. We say that an application is a *SOA* system if it is partitioned into services that interact directly, independent of context.

The concepts in the above two definitions are based on the definitions provided by [7] and [8]. In [7] a service is defined as "a set of functionality provided by one entity for the use of others". We feel that it is important to emphasize direct interaction to avoid confusion with technologies such as Common Gateway Interface (CGI) (accessed through web servers), and Components that interact through specific component frameworks.

B. Contribution

Our major contribution is tracing the evolution of SOA in relation to its drivers and how they are addressed at the architecture level. We also show the exact features inherited from its predecessors and corresponding enhancements on each of the features. In this way, we facilitate understanding of SOA and foster its proper usage to leverage its benefits.

The rest of the paper is organised as follows. In section II, we discuss work that deals with architecture issues of SOA. Section III gives a brief review of software architectures and section IV shows how SOA has evolved from its predecessors, inherited features and enhancements. In section V, we give a conclusion and future work.

II. Related Work

We highlight work that attempts to address the understanding of SOA in terms of software architecture. Stal [2] suggests the use of architecture patterns and blueprints to explain the architecture principles of SOA. The work bases on what he calls driving forces to define a model for the service oriented context. The work addresses the architecture patterns necessary to address the driving forces. However, he restricts his work to implementation and design patterns. which clarifies most implementation issues but does not do much to facilitate the understanding of SOA in mannar independent of implementation and design. Our effort is aimed at explaining SOAs by stressing its architecture features in relation to the predecessors.

In [7], a comparison of service oriented and Distributed Object Architectures is given. We agree with their view that: "while superficially similar, the two approaches exhibit a number of subtle differences that, taken together, lead to significant differences in terms of their large-scale software engineering properties such as the granularity of service, ease of composition and differentiation properties that have a significant impact on the design and evolution of enterprise-scale systems"[7]. We however, move a step further to show how specific features have been enhanced to make SOA an attractive appraoch.

III. Software Architectures

While a variety of definitions of the term software architecture have been suggested, this paper will use the definition first suggested by [9] who saw it as the structure of a system, which comprised of components or building blocks, the externally visible properties of those components and the relationships among them. Our view of software architectures is inspired by the definition provided by [10] who define software architecture as the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.

We note that whereas SOA does not represent a specific architecture, it did not come out of vacuum. It is an attempt to address particular architecture concerns. It is therefore important to establish the exact concerns and how they are addressed at the architecture level. In light of the above definitions of software architecture, SOA and in line with the theme of this paper, we focus on the structures and the interactions that are specified by SOA. Our particular interest is in those structures and interactions that make SOA a novel approach.

A. Architecture building blocks in SOA

SOA is described as a framework, as a paradigm [8], as a collection of services [11] and [7], plus several other definitions. What is clear from these definitions is that they do not represent the same thing. We would like to distinguish two uses of the term SOA. In one context, SOA is used to refer to an abstract architecture (architecture that do not represent concrete systems) [8]. However, SOA has another meaning to it in which it is used to refer to concrete systems (running applications) [11] and [7]. In this context, just like CORBA, SOA does not represent architecture of a specific system just like CORBA. Definitions such the *framework* and *paradigm* refer to this dimension of SOA and are common in the industry and practitioners. As a framework, paradigm or way of building architectures, SOA describes how to specify services and facilitate the actual interaction between services.

In the second context, definitions such as *collection of services* aim at answering the question; when is a system service oriented? One would expect the first context to imply the other. In other words, if SOA is a framework or paradigm, systems designed as per this framework or paradigm should have a service oriented architecture. Unfortunately this is not the case because some key principles in the SOA framework are non-architectural (such as how to 'publish' and 'find' services). Just like component repositories do not contribute to the software architecture, the way services are *published* and *discovered* is not part of a running SOA.

In terms of software architecture, the architecture building blocks of SOA are services [12]. Services are the computational elements exposed in a manner that allows flexible and uniform access. The elements provide a consistent way of connecting to each other through well defined interfaces. The services then interact through message exchanges. Coordinating interaction among services is in fact equivalent to creating connectors. Therefore, SOAs separate coordination from composition.

B. Component Models

It has been suggested that there is a strong relationship between component models and SOA [5],[6]and [4]. Thus, for purposes of completeness, we provide a brief review of components and component models. We adapt the sofware component in [13] as a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. The basic architecture of such components is defined by a corresponding component model, such as Enterprise Java Beans (EJB) [14], Common Object Model (COM) [15], or Common Component Model (CCM) [16].

Components are the major building blocks for Software Architecture [17]. Therefore, components have their own internal architectures. Component based development proceeds by composing software systems from pre-fabricated components [18] and [6]. Brown and Wallnau [19], suggest that, the initial stage involves the selection of components that are likely to satisfy the main system requirements and view component based development to be primarily an assembly and integration process.

The process of assembly and integration is based on the expressiveness of the Component Model in use. Expressive Component Models make it easy to architect systems. In [20], they asserts that, a typical component-based system architecture comprises a set of components that have been purposefully designed and structured to ensure that they fit. Therefore, using components from a specific component model aids reasoning about the desired system. Component Based Software Systems often constitute components owned by the same organisation using the same Component Model.

Components and connectors have been for a long time considered as the only basic first class architecture elements. As such, they are always associated with software architectures of systems. We are currently witnessing increased interest in service based systems with service being considered as first class architecture elements. However, there is no comparison between services and components or other existing architectural elements. The comparison can not be less hard in the absence of clear definition and structure of a service. Consequently, the relationship between services and components has not been well articulated. This would help to clarify the benefits of service oriented systems and provide guidelines on how to benefit from both components and services.

IV. Evolution of SOA from its Predecessors

The emergence of SOA is described using figure 1. The aim is to put SOA in context basing on its drivers (concerns) , architecture properties, architecture specification and instances.

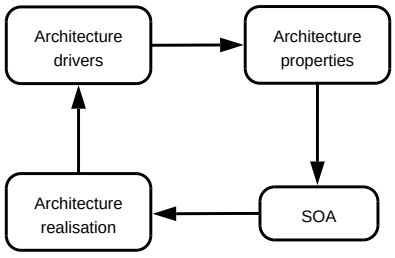

Fig. 1. showing relation between SOA concerns (drivers), architecture features and architecture

Like any other architecture, SOA emerged due to limitations in existing approaches. These limitations create a need for a new architecture approach and become the driving force behind its success. In the case of SOA, special purpose concerns (drivers) are extensibility, flexibility, connectivity and interoperability [21]. Whereas attempts to address some of these concerns are evident in Component Models such as CORBA [22], their treatment is as far as the global objective of the Models. Globally, Components and Component Based Software Engineering (CBSE) methodologies focus much on enabling distributed computing with little attention to enterprise computing and associated dynamics.

SOA is a distributed computing architecture [12], [23] and [8]. Remote procedure Call (RPC), CORBA, Distributed Computing Component Model (DCOM), and Remote Method Invocation (RMI) are (were) the most widely used and well known distributed computing architectures. Each appraoch was designed with specific concerns in mind and is less effective in other situations. Typical distributed computing approaches put much emphasis on the technical details and requirements for the composition and the distribution of the components. Although this is a core necessity, more attention is needed on the global picture of the resulting systems and how they fit into the dynamic enterprise computing environment. This lack of attention to enterprise issues is supported by the popularity of commercial middleware products that try to fill in the missing details to make CBSE and component models suitable for the enterprise environment.

Specifically, the emergence and specification of SOA is shaped by the global picture of an enterprise computing environment. An environment where flexible, open and interoperable architectures are preferred. To provide such an environment, SOA addresses the concerns at the architecture level by refining and enhancing the features from its predecessors. Surprisingly, the combined refinement and enhancement on different features has resulted into a novel architecture approach in SOA.

A. The strength of SOA

In their study of Interfaces and Objects, [24] assert that current software applications can be separated into a basic concern and a number of special purpose concerns. They argue that: “The basic concern is represented by the fundamental computational algorithms that provide the essential functionality relevant to an application domain and the special purpose concerns relate to other software issues, such as user interface presentation, control, timing, and distribution”. Therefore, architectures can differ in terms of basic concerns or special concerns. At the basic concern level, we look at the structure of the computational elements.

However, SOA is an abstract architecture and does not specify concrete computational algorithms, but specifies the type of architecture elements to be used. Abstract architectures therefore differ in terms of the elements they describe and the interactions to be used. Therefore, to understand the difference and similarities between SOA and its predecessors, a comparison is needed at two levels : *i)* basic concerns and *ii)* how they interact. A one-to-one structural comparison of Web services and CORBA [5] does not yield much. It only reveals philosophical differences that translate to efficient interaction and only contribute to the overall architecture of the resulting systems. Wang and Fung[11] list characteristics

of SOA which show that SOA emphasises interaction rather than structure. Interactions manage the behaviour of elements and therefore have substantial influence on the overall system. By focusing on interaction, SOAs mitigate the limitations of the predecessors.

Considering [7], [11] and [24], we conclude that SOA's strength does not lie in its computational elements but in the way it enables and handles their interaction. It focuses on the describing externally observable behaviour [25]. SOA does not emphasise computational elements, and therefore does not derive its uniqueness from structure. SOA emphasises what [24] refers to as special purpose concerns. Special purpose concerns in this case have been described in section IV.

In a nutshell SOA does not introduce new features but enhances existing concepts. What really matters is the emphasis on specific features that address special purpose concerns which translate into improved software architectures. This puts to rest the debate whether a service is a component or vice versa. A service can be constructed from scratch or from components (just like from legacy systems). The latter does not imply that we get an aggregate component or aggregate legacy system, because SOAs strength is not in its internal structure but in its externally visible behaviour. The overall addition of SOA to its predecessors is the treatment of enterprise computing concerns at the architecture level.

B. Features from Predecessors

We identify abstraction, composability, separation of concerns and extended interfaces as the key architecture features that are enhanced in SOA to satisfy its drivers. The features identified here are those intentionally designed (can be improved) at the architecture level to satisfy the SOA concerns. In addition, these features combine to produce other desirable properties at the global level. Features such as loose coupling, autonomy are a consequence of deliberate designs at architecture level. These features already exist in the SOA predecessors but have been enhanced to make SOA unique. We discuss the features from the point of view of the predecessors.

1) Abstraction: : SOA extends the concept of abstraction and encapsulation. Unlike in OO which hides internal details (data and methods), SOA aims at hiding all causes of integration problems. Stal [26] lists the causes of heterogeneity as "network technologies, devices, and OSs; middleware solutions and communication paradigms; programming languages; services and interface technologies; domains and architectures; and data and document formats."

SOA encapsulates a specific set of discrepancies in its domain of application. By hiding these differences, services within the service oriented architecture can be accessed seamlessly. In terms of architecture, the differences are typically hidden by using a common set of standards. SOA instances are designed for specific domains thereby hiding specific causes of discrepancies. This is supported by Web services which is an SOA instance and targets the Internet as the mode of delivery, and therefore requires standards that hide heterogeneity in platforms. A simple SOA that is perhaps part of an operating system such as accessing devices will require standards that present the devices in the same way. A typical example is UNIX where everything is a file. In terms of architecture, it implies that standards play a critical role. Through use of standards, protocols and common vocabulary, the differences in participants are abstracted thereby allowing all the participants to appear uniform.

Generally, SOA does not eliminate the heterogeneity, but simply hides its causes. The systems and technologies in the background remain heterogeneous [26] but their interfaces and collaborations are standardized. This is a core concept of SOA and the choice of interface specification and standards is left to different instances of SOA. The choice and design of the standards must be carried out with care lest they will be a bottleneck. They must be simple to use and comprehend without compromising the efficiency. Simple standards tend to be very verbose while short standards are normally hard to understand. A key to this is a compromise between the two extremes. According to [27], the protocols and standards that enable technology abstraction should be suitable and intuitive for application developers that develop and maintain applications in different technology domains.

Prior studies have noted the importance of uniform access in SOA [28], [7]. As [28] argue : "a key goal of WS framework is to produce a common representation of applications which use diverse communication protocols and interaction models while at the same time enabling Web services to take advantage of more efficient protocols when they are available at both ends of the interaction".

SOA therefore enhances abstraction and encapsulation to a level higher than what OOA and CBA provide. As an architecture, SOA is intentionally designed with a set of technologies and standards to provide the required level of abstraction.

2) Composability: : The concept of composition is well articulated in component models and applied in CBSE to build bigger systems. SOA carries forward the belief that complex systems can be built from smaller elements by means of compositions. However, composability in CBSE is constrained by restrictions in component models. Existing literature [13],[29], [20] and [7] shows that components rely on a component model within which to operate. Component models are in fact the major limitation to interoperability, with existing component models such as DCOM, CORBA and EJB not able to easily interoperate. This makes component based system not suitable for enterprise computing where we can not control the component models used by partners we wish to interoperate with.

SOA waives the restrictions imposed by component models. This is a clear enhancement on the composability feature. In SOA, there is no requirement for a specific internal structure, no component model semantics, no containers and application servers. This relaxation fosters software agility where software systems change frequently and gracefully.

3) Extended Interfaces: : In addition to syntactic specification (collection of function signatures) as is the situation for

interfaces in current commercial component frameworks [21], services need to specify how they are to be used (i.e. interaction patterns). These interaction patterns give an aggregate behaviour of the service in the form of protocols. Thus, composition of services is based on the interaction protocols rather than individual operations as in components.

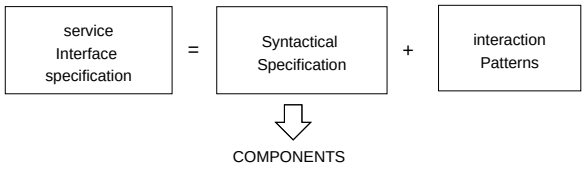

Fig. 2. relation between component and service interfaces

According to [30], interactions mimic business environment. In this respect, a service is a component augmented with business meaning using interaction patterns. With the addition of interaction patterns to syntactical specification, the following benefits are leveraged

- ◇ The autonomy of the services is increased since services dictate how they are to be used. Valid usages are those that do not violate the interaction patterns specified by the service.
- ◇ Loose coupling is extended to a level beyond the syntactical specification of services. The services interact and compose based on the interaction patterns
- ◇ There is an added avenue for standardization. Similar services from different venders in the same vertical domain (such as air lines, procurement) may provide standardized interaction patterns thereby improving dynamic reconfigurations in cases of failure.
- ◇ Improved agility and more reuse. New interaction patterns can be defined for the same services there by providing an additional avenue for reuse.
- ◇ The interaction patterns provide a basis for validation and verification of compositions. In addition, the need for explicit business process languages is greatly reduced and provided within the service interaction patterns.

4) Separation of Concerns: This principle states that a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity, and to achieve the required engineering quality factors such as robustness, adaptability and reusability [31]. In SOA the major concerns are integration of autonomous systems which is directly influenced by the way services are exposed and interact. Thus, the critical starting point addressed by SOA is the way systems are exposed and consumed. SOA requires that systems are accessed through platform independent interfaces.

In CBSE and component models, we have clear separation of implementation issues from the interfaces. The components internal structures are separated from the way the component is to be accessed. This concept is well extended in SOA by providing service description in addition to interface specification. Take a case of Web services and CORBA. An Interface Description Language (IDL) in CORBA may be compared to WSDL, but the IDL is no more than the interface specification. Its purpose is to provide a common interface specification language instead of using multiple languages. Web services use the WSDL as sharable document and basis for communication between services while CORBA shares stubs generated from IDL in order to enforce communication. In a sense, web services allow more exchange of detailed description information which enables explicit description and separation of issues such as QoS and transactions. In addition, more separation of concerns can be seen at the UDDI where business related concerns are addressed for purposes of discovery and subsequent interaction

This enhancement on separation of concerns at different levels provides an efficient way of separating technical requirement for composition and distribution from issues at the system and enterprise level. However, a key concept in Object Oriented Programming (OOP) that is not necessary in SOA is inheritance mainly because SOA is not a programming model and services are self contained, independent entities whose internal details are invisible. Services do not need to inherit any functionality since they can include the functionality through standardized service interactions.

C. Additional features of SOA

We discuss features that are not directly traced to any of the predecessors. They are as result of a combination of the different features and associated enhancements.

1) Autonomy: In an enterprise environment, organizations own systems and interact by providing services to each other. Services are expected to be developed and owned by different organizations or individuals. SOA is therefore about facilitating collaborations of independent services into coherent and meaningful systems. The fact that services are autonomous in every sense ranging from development to ownership, they are bound to be incompatible in many aspects. SOA therefore involves reconciling disparate services into a seamless application. To promote reuse, the functionality provided by an autonomous service should be functionally cohesive. However, this may not be possible if services are designed to target end user requirements and business logic that often require a collection of unrelated functionalities. For this reason, a layered architecture is advocated where the first layers provide basic services that serve as blocks for building the next layers that successively approximate to business processes. The lower services may handle issues like storage and communication.

2) Loose coupling: Loose coupling is the most pronounced architecture property associated with SOA. Of course loose coupling is not really a new concept unique to SOA. It exists in CBSE, except SOA is designed to be more loosely coupled. SOA achieves more loose coupling through its emphasis on platform neutral interfaces, platform neutral descriptions, message based interactions and self-contained services. In tightly coupled systems, architecture elements are specifically designed for each other, while in loosely coupled systems, architecture elements are designed to interoperate. At the

architecture level, SOA achieves more loosely coupled systems through the enhancements on the features discussed above.

D. Critical differences

We specify some critical differences at both the service level and the system level. We explain how the relaxation on some of the features impacts on the overall architecture of the resulting system.

1) Deployment Environment: Unlike CBSE where components are required to run in the new integration environment, services are used on *as is* basis. They are consumed from their natural environment. This places less requirements for service specification because the focus is only on interaction with other services and not with the environment. Consequently, the integration problem is eased since services exclude the need to support a new running environment.

2) Required operations versus provides operations: According to [32], to specify a component, it is necessary to define it in terms of the operations it provides and the operations it requires. Services do not specify the required operations; they only specify what they provide. Ideally, such information (required operations) is not necessary since services are consumed wherever they are located and therefore already have all that is required. Services may rely on other services, but to the consumer, this is transparent. .

V. Conclusion

We have shown the relation between SOA and concerns (drivers), architecture level features and how they relate to SOA predecessors. We have described the exact enhancements on the architecture features inherited from the predecessors. In so doing, we have clarified most architecture issues surrounding SOA. Through our architecture analysis, we have advanced the view that *SOA's uniqueness and strength does not lie in its computational elements but lies in the way it enables and handles their interaction*. In our future work, we intend to provide more backup on this view, by looking at the contribution of interactions (connectors) to software architectures. Therefore, we recommend that future research in architecture issues relating to SOA should focus on optimizing this view.

References

[1] W. T. Tsai, "Service-oriented system engineering: a new paradigm." in *SOSE 2005*, 2005, pp. 3– 6.

[2] M. Stal, "Using architectural patterns and blueprints for service-oriented architecture." *IEEE Software*, vol. 23, no. 2, pp. 54–61, 2006.

[3] S. Loughran and E. Smith, "Rethinking the java soap stack," in *IEEE International Conference on Web Services (ICWS)*, Orlando, Frolida, USA, July 2005.

[4] I. Crnkovic and V. Jamwal, "Components and services session report." in *WICSA*, 2005, pp. 269–271.

[5] A. Gokhale, B. Kumar, and A. Sahuguet, "Reinventing the wheel? corba vs. web services," in *Proceedings of The Eleventh International World Wide Web conference (WWW2002)*, Honolulu, Hawaii, 2002.

[6] C. A. Szyperski, "Component technology - what, where, and how?." in *ICSE*, 2003, pp. 684–693.

[7] S. Baker and S. Dobson, "Comparing service-oriented and distributed object architectures." in *OTM Conferences (1)*, 2005, pp. 631–645.

[8] OASIS, "Reference model for service oriented architecture draft 1.0," 7 February 2006.

[9] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. New York: Addison-Wesley, 2003.

[10] IEEE-1471, "Recommended practice for architectural description of software intensive systems," 2000, iEEE Standard 1471-2000.

[11] G. Wang and C. K. Fung, "Architecture paradigms and their influences and impacts on component-based software systems." in *HICSS*, 2004.

[12] M. P. Papazoglou, "Service -oriented computing: Concepts, characteristics and directions," in *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISEâ03)*. IEEE, 2003.

[13] G. T. Heinman and W. T. Councill, *Compoment Based Software Engineering: Putting the pieces together*. Addson Wisley, 2001.

[14] L. G. DeMichiel, L. U. Yalcinalp, and S. Krishman., *Enterprise JavaBeans Specifications Version 2.0,*, 2001.

[15] D. Box, *Essential COM*. Adison-Wesly, 1998.

[16] OMG, "Corba Compponet Model,V3.0, 2002." *http://ww.omg.org/technology/documents/formal/components.htm*, 2004.

[17] M. C. Oussalah, A. Smeda, and T. Khammaci, "Software connectors reuse in component-based systems." in *IRI*, 2003, pp. 543–550.

[18] A. W. Brown and K. C. Wallnau, "The current state of cbse," *IEEE Software*, vol. 5, no. 15, 1998.

[19] A. Brown and K. Wallnau, "Engineering of component-based systems." in *ICECCS*, 1996, pp. 414–422.

[20] K. Gerald, J. Hutchinson, and B. Bloin, "Compose: A method for formulating and architecting service-based systems," in *Service-Oriented Software System Engineering: Challenges And Practices*. Idea Group Inc, 2004.

[21] R. Perrey and M. Lycett, "Service oriented architecture," in *SAINT Workshops*, 2003, pp. 116– 119.

[22] OMG, "The common object request broker: Architecture and specification (corba) rev. 2.6." in *Object Management Group (OMG),retrieved on 12th December 2004 from: http://www.omg.org*, 2001.

[23] F. Chen, S. Li, and W. C.-C. Chu, "Feature analysis for service-oriented reengineering." in *APSEC*, 2005.

[24] P. S. C. Alencar, D. D. Cowan, and C. J. P. de Lucena, "A logical theory of interfaces and objects." *IEEE Trans. Software Eng.*, vol. 28, no. 6, pp. 548–575, 2002.

[25] N. K. Diakov and F. Arbab, "Compositional construction of web services using reo." in *WSMAI*, 2004, pp. 49–58.

[26] M. Stal, "Web services: Beyond component-based computing," *Communications of the ACM*, vol. 45, no. 10, pp. 71–76, 2002.

[27] J. P. A. Almeida, L. F. Pires, and M. J. van Sinderen, "Web services and seamless interoperability," in *ECOOP 2003 European Workshop on Object Orientation and Web Services*, Darmstadt, Germany, July 21 2003.

[28] F. Curbera, N. W, and S. Weerawarana, "Web services. why and how," *Workshop on object-Oriented Web Services OOPSLA, Tampa, Florida,USA*, vol. 12, pp. 591–613, 2001.

[29] K.-K. Lau and Z. Wang, "A taxonomy of software component models." in *EUROMICRO-SEAA*, 2005, pp. 88–95.

[30] M. E. Maximilien and M. P. Singh, "Towards web service interaction style," in *IEEE SCC*, 2005, pp. 147–154.

[31] M. Aksit, B. Tekinerdogan, and L. Bergmans, "The six concerns for separation of concerns," in *ECOOP-Workshop on Advanced Separation of concerns*, Budapest, 2001.

[32] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall., 1996.