Software Service Engineering: Tenets and Challenges

Willem-Jan van den Heuvel[1], Olaf Zimmermann[2], Frank Leymann[3],
Patricia Lago[4], Ina Schieferdecker[5], Uwe Zdun[6], and Paris Avgeriou[7]
[1]*Tilburg University,* [2]*IBM Zurich Research Lab,* [3]*Stuttgart University,*
[4]*VU University Amsterdam,* [5]*Fraunhofer Institute,*
[6]*Vienna University of Technology,* [7]*University of Groningen*

Abstract

Service-Oriented Architecture (SOA) constitutes a modern, standards-based and technology-independent paradigm and architectural style for distributed enterprise computing. The SOA style promotes the publishing, discovery, and binding of loosely-coupled, network-accessible software services. With SOA systems operating in distributed and heterogeneous execution environments, the engineers of such systems are confined by the limits of traditional software engineering. In this position paper, we scrutinize the fundamental tenets underpinning the development and maintenance of SOA systems. In particular, we introduce software service engineering as an emerging discipline that entails a departure from traditional software engineering disciplines, embracing the 'open world assumption'. We characterize software service engineering via seven defining tenets. Lastly, we survey related research challenges.

1. Introduction

Service Oriented Architecture (SOA) is rapidly emerging as a premier distributed computing paradigm for developing, integrating, and maintaining enterprise applications [8]. Many organizations are now in their early use of SOA, and assume that they can simply apply principles and techniques from pre-existing software engineering paradigms such as Object Orientation (OO) [5] or Component-Based Development (CBD) [2], or the traditional architecting approaches (e.g., based on component-and-connector views) to engineer services. These principles and techniques are independent of any architectural style. SOA-enabled applications operate in distributed, non-deterministic, unpredictable, and highly dynamic heterogeneous execution environments; hence, SOA engineers quickly encounter the limits of such traditional software engineering paradigms, which do not provide any style-specific advice. Moreover, SOA confines itself to a rather simple reference model for development and deployment, defining basic roles such as service consumer, service provider, and service broker. It is left up to the discretion of the engineers how to construct software service applications in this rather generic model.

Our ultimate objective is to scrutinize the viability of existing engineering paradigms for developing and evolving software service-based applications, including CBD and OO, and to explore their shortcomings. In this particular paper we investigate the distinguishing characteristics of an emerging engineering discipline for development and maintenance of SOA-enabled applications, which we call *Software Service Engineering (SSE)*. We introduce the key SSE tenets. Furthermore, we landscape the key challenges for establishing SSE as a discipline.

The research that is presented herein has been conducted adopting a research approach combining our background from literature surveys, case studies, and industrial best practices with brainstorming sessions involving representatives of several communities – including researchers and practitioners from the domain of software engineering, software patterns, SOA, and method engineering.

The remainder of the paper is organized in the following way. Section 2 presents background information on SOA as an architectural style and introduces an example. Section 3 identifies SSE tenets. Section 4 derives SSE research challenges from the tenets. Section 5 provides a synthesis and gives an outlook to future work.

2. Principles and Patterns in SOA Design

SOA is an architectural style based on principles such as location, protocol, format, and technology platform independence and patterns including *Service Contract*, *Service Composition, Enterprise Service Bus (ESB)*, [6], and *Service Registry*. SOA allows service engineers to (re-)organize and (re-)deploy business processes, functional components, and information

978-1-4244-3716-0/09/$25.00 © 2009 IEEE

assets as business-aligned, loosely-coupled, and autonomous *software services*. SOA is unique in that it combines elements from various related, yet up to now largely isolated disciplines such as business process modeling and management, software architecture, CBD, OO, Enterprise Application Integration (EAI), distributed computing, and systems management.

First and foremost, SOA design is architecture design: software architects on SOA projects are responsible for defining the architecturally significant requirements (ASRs) during *architectural analysis* [3]. ASRs include functional requirements typically captured in stories, use cases, or business process models, but also non-functional requirements such as software quality attributes. Subsequently, architects make design decisions to satisfy the ASRs during *architectural synthesis* [3]. A number of architectural views covering different design aspects are chosen and populated iteratively and incrementally during this activity. During the activity of *architectural evaluation* [3], the architects ensure that their decisions satisfy the ASRs in an optimal (or at least good enough) way. Furthermore, architects lead project teams via coaching and review activities, and manage the relationships with external stakeholders on the technical level. All these activities and interactions influence each other.

At the early elaboration stages of architectural synthesis, the conceptual architectures of SOA-based systems can be designed in a straightforward manner: they are variations of logically layered two- or three-tier client-server architectures. In such architectures, enterprise integration patterns are used to let *consumers* and *providers* of software services exchange messages via the ESB; workflow concepts provide one way of composing services [6]. A service registry serves as directory of service providers available to respond to service consumer requests. The service request and response message formats are defined in the service contract. Architects are also concerned with the design and configuration of middleware such as ESBs (responsible for request routing, adaptation, and mediation), workflow and process orchestration engines (facilitating service composition), and service registries (supporting provider lookup). Individual service consumers and providers are designed, developed, and then deployed into such *SOA infrastructures*.

Solutions to these and numerous other design issues have been successfully codified into *design patterns* and *architectural patterns* by the software patterns community. However, the application of style-specific patterns in the daily practice of architects during SOA design has not been particularly successful to date.

Example. To give an example for SOA design, Figure 1 shows a traditional application landscape:

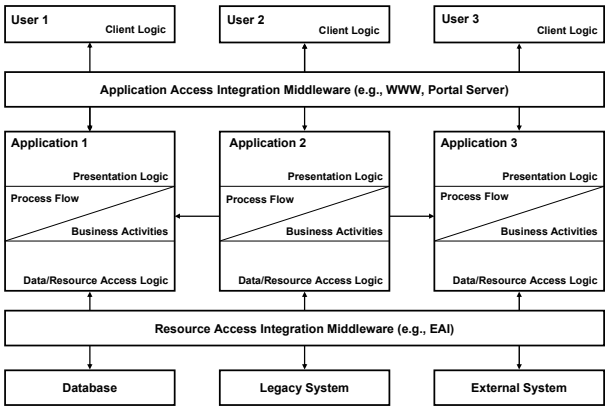

Figure 1. Logically layered applications

Three applications serve three user clients over an application access integration middleware, e.g., a portal server which can be reached over HTTP and the World-Wide Web (WWW). The applications are logically layered, conforming to the layers pattern and a particular layering scheme for enterprise applications proposed in the literature. Each of these applications is statically assembled and deployed to a runtime platform (i.e., hardware, operating system, and container). An example is the deployment of the three applications (in the form of .war/.ear) files to a Java Enterprise Edition (JEE) Web application server which may run on a Linux server. Via EAI middleware, the three applications integrate a database, a legacy system, and an external system (backend systems).

Figure 2 shows the same three user clients and three backend systems. The three applications are refactored into an SOA comprising two *service consumption assemblies* and a *service provider pool*.

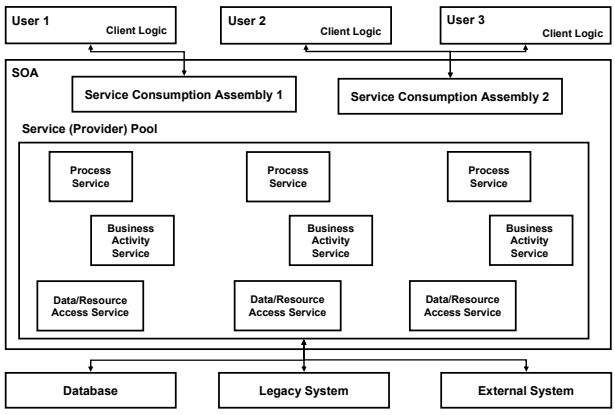

Figure 2. SOA with service pool

The service provider pool contains *process services*, e.g., business processes implemented in the Business Process Execution Language (BPEL), *business activity services* (e.g., wrapped Java components), and *data/resource access services* (e.g., provided by SQL adapters and connectors). The service consumption assemblies can take several forms, e.g., traditional application or Web 2.0 mash up. Service consumers and providers communicate over the ESB, often using SOAP as a message exchange protocol (note that the ESB is not shown in the diagram for reasons of space). Each service provider exposes a service contract.

We will return to this example and reveal more SOA design details in the next section on SSE tenets.

3. SSE Tenets

Software architects and designers cannot be expected to embark on large-scale SOA projects without relying on sound defining principles underpinning the methods, techniques, and tools required for SOA application and infrastructure design, development, and maintenance. Without such principles, which we refer to as *SSE tenets*, it cannot be guaranteed that the developed software systems meet the SOA principles which ensure that services are loosely coupled and specified via clean interfaces that are geared towards flexible and dynamic (re-)composition.

During a Schloss Dagstuhl seminar in January 2009 [1], we gathered such distinguishing SSE tenets. This was achieved by facilitating two half-day working sessions. During the first session, candidate SSE tenets were identified and analyzed in two groups of about 25 participants. The second session was a plenary session during which the two proposed lists were correlated, integrated, and consolidated. Note that due to reasons of space, we have not included transcripts of these discussions. The discussions were initiated with the help of a list of potential tenets that had been distilled in a literature survey that analyzed the tenets underpinning OO and CBD (including seminal works such as [9], [10], [11], and [12]), but also input from other fields such as telecommunication services [13], networking [14] and testing [15]. The following unordered list of SSE tenets was identified:

1. Technical federation. SSE has to cater for service-enabled software applications that are logically and physically distributed. Message-based communication via an ESB permits and encourages *asynchronous, non-blocking interactions* between service consumers and service providers: A provider reacts on an incoming request message (which can be seen as an event), by default not knowing of and not making any assumptions about the originator of the request. The provider may, but does not have to, respond with a response message, which may be sent over a separate communication channel.

In the example from Section 2, asynchronous communication is suited for the channel between a data/resource access service and a legacy system which is located in a different location and slow and unreliable when responding to requests. As a consequence, process and business activity service execution (which we assume to have invoked the data/resource access service) is not blocked while waiting for the response from the legacy system (referred to as the send-and-forget principle). The request and response messages are queued by the ESB (following the store-and-forward principle). This is an example of loose coupling in the time dimension; it makes the SOA design flexible and reliable. This increase in flexibility and reliability, however, comes at a price: The amount of design and development increases significantly as the service consumers have to keep track of open requests, correlate incoming responses to requests, handle timeouts, resequence messages arriving out of order, and so forth.

As of today, distribution is typically either realized with EAI middleware such as message brokers or with remote procedure (or object) invocations as supported by application servers. Both integration styles continue to be relevant for SSE; if asynchronous, message-based integration is used, service providers and consumers as message endpoints can not make any assumptions about the technical nature or lifecycle of their communication partners. Fewer assumptions mean more design work: SSE has to provide architects and other *software service engineers* with concepts, languages, methods, and tools that help them manage this increased technical complexity, sometimes referred to as “programming without a call stack” [6].

2. Dynamism (virtualization). A key tenet of SSE is dynamism regarding the services that are aggregated into service compositions at runtime via *late binding* (forming agile *service networks*), as well as the highly volatile context in which such services operate. First, dynamism implies that SSE methods, techniques, and tools have to deal with emergent properties and behavior of complex service networks, which may be comprised of thousands of independent – yet cooperating – services. In fact, complex and emergent behaviors includes both technical issues such as performance and security, as well as business issues including profitability, return-on-investment, and indices of value creation. Dynamism puts requirements on virtually all elements of an SOA platform, ranging

from the ESB messaging infrastructure to composition engines. Loose coupling and late binding constitute two key principles for increasing the adaptability of service applications, accommodating the flexible, dynamic (re-)composition and (re-)configuration of services in a network. This flexibility and dynamicity may lead to a large number of consumers per provider. SSE also has to accommodate various styles of service composition, fostering user-friendly enterprise service mash-ups as well as heavy-weight compositions of industry-strength enterprise applications.

In our example, the service consumption assemblies may look up available process services at runtime, depending on the content of incoming user requests (e.g., premium vs. regular customer) and the current environmental context (e.g., load). Similarly, a process service may pick a different business activity service based on current processing state and required Quality of Service (QoS). The SOA can be set up in such a way that multiple service providers implementing the same service contract are provisioned. The actual provider can be dynamically selected at runtime based on certain policies, e.g., content-, QoS, and load-based routing; it is hidden from the consumer (*service virtualization*). In such a setting, the service pool is a single deployment entity shared by all user clients (unlike in traditional application development).

The composition (assembly) of programming language artifacts such as modules into higher order constructs is a known concept; for instance, patterns such as pipes-and-filters and model-view-controller promote it. The SOA style adds *workflow concept*s as an additional conceptual means of process service composition.

The runtime lookup of components via a naming and directory service also is well understood and supported in existing middleware, e.g., in the CORBA directory service and JNDI defined in JEE. Lookups in an SOA differ from those in OO and CBD in that the unit of lookup is not a remote object reference accessible via a home and/or remote interface (such as in CORBA or in JEE Enterprise JavaBeans), but a network-accessible service with always-on semantics. In the absence of a formal specification or a universally agreed industry reference model for SOA, the line between services and components is somewhat blurred. In the literature, we find different positions such as: a) services are equivalent to components; b) services are a superior paradigm making components obsolete; and c) services and components complement each other, i.e., components can be used to implement and provide services. Future SSE assets (e.g., methods, tools) must make their own interpretation of the two terms and their relation explicit. Service registries in an SOA must support diverse lookup scenarios such as by name as in existing middleware, but also by contract type, by business taxonomy, and by QoS policy.

3. Organizational federation. SSE should be shaped around the doctrine stating that development and maintenance are often conducted in distributed organizational units, possibly involving multiple lines of business, other enterprises, and government institutions. Development and maintenance of applications becomes a collaborative effort, implying that design, coding, deployment, etc. occur in networks of collaborative service clients and providers. The presence of a central governance body such as an architecture board can not be assumed. Organizational federation requires sound distributed governance mechanisms, accommodating the individual needs of the various stakeholders, which often stem from organization-specific constraints or governmental rules and legislations. Organizational federation may adopt a range of coordination mechanisms, ranging from a classical central control system to a decentralized structure, relying on mechanisms such as *service markets* and contracts.

In the example from Section 2, each of the three traditional applications is typically developed by a separate project team (which may be geographically distributed, but are managed centrally). In the SOA case, each service provider in the service pool and each service composition assembly may be developed autonomously by a different unit or legal entity.

Organizational federation in itself is not new. For instance, the Eclipse project team is spread over several locations in Europe and North America. It employs a self-governing, agile engineering process, the Eclipse Way. In an SOA design context, the strong emphasis on technical federation, dynamism, and heterogeneity (as expressed as separate SSE tenets) makes the need for supporting concepts particularly relevant. The reuse of shared services required a sound approach to service ownership and lifecycle management. For instance, a particular difficulty is the versioning of shared services that are used by multiple consumers in different organizational domains.

4. Explicit boundaries (contracts). Services developed with SSE methods or tools have to be endowed with clear boundaries, which are made explicit in the form of contracts. In particular, SSE has to provide service contracts that capture goals and constraints (pre- and post-conditions and invariants), capitalizing Meyer's classical design-by-contract principle [16]. An intrinsic part of the service contract entails the service interface that specifies the messages a service understands and the service endpoints that are

available (*structural contract*). Enriching the service interfaces with additional semantic information such as scenarios or behaviors allows a more robust and stable service composition (*behavioral contract*). In addition, given the highly distributed and volatile nature of service applications, there is a clear need to amend service contracts with *Service Level Agreements* (SLAs) between service consumers and service providers which allow service consumers to express the expected and service providers to specify the available QoS (*policy contracts*). Machine-readable contracts allow the ESB and service composition middleware to support other SSE tenets such as dynamism and business-IT alignment.

In the example from Section 2, each of the rectangles representing an architectural component may expose such contract (both in the traditional application landscape and in the SOA).

Traditional software engineering emphasizes the need for explicit boundaries and contracts; the tenet is one of the key elements of OO and CBD. Traditionally, the main focus has been the structural contract as seen from the provider perspective. In SSE, a particular challenge is the absence of a single platform model defining a call stack and storage model: It is not possible to look inside a service implementation to verify postconditions and invariants. However, SSE can use the principles of *built-in testing* allowing for services to contain their own test specification and enabling their run-time verification [18].

5. Heterogeneity. Any SSE concept, method, or tool has to embrace heterogeneity of SOA applications and infrastructure and the context in which these architecture elements operate. Just like dynamism, heterogeneity impacts all phases of the service development and maintenance lifecycle, posing restrictions on how software services can be designed, developed, deployed, and evolved over time. Note that in contrast to current practices, no assumptions can be made about the system's programming, execution, and management context before, during, or after deployment. SSE has to deal with services that may be deployed on various runtime platforms – including mobile devices, compute clouds, and legacy systems – and have been developed under various programming paradigms – including OO and CBD.

In our SOA example, the user client may be a PHP script; the process services might be executable BPEL or BPEL^{light} process hosted by a BPM engine; the business activity services might be implemented as Java or C# components; the data/resource access services might be provided by an application server or by ESB adapters; the database might be a relational database; the legacy system might be a software package (e.g., from an enterprise resource planning vendor) or a homegrown COBOL program running on an mainframe computer; the external system might be a RESTful service available on the Internet; etc.

Integrating heterogeneous application landscapes is the objective of EAI (and, to some extent, multi-platform CBD platforms such as CORBA). SSE has to integrate proven principles and patterns from these fields and assets. The principles and patterns have to be refined to take advantage of the SOA style-defining concepts such as services, service composition into processes, and asynchronous ESB messaging.

6. Business-IT alignment. SSE embraces a new style of development assuming that SOA applications can be systemically and routinely (re-) mapped to the business processes they realize, and vice versa. This suggests the need for a unification of concepts, models, methods, and techniques from Business Process Management (BPM) and software engineering to ensure that these applications do not only meet system-level QoS criteria, but also perform as specified in certain business process-level Key Performance Indicators (KPIs). The monitoring of such indicators is referred to as Business Activity Monitoring (BAM) nowadays.

In the example introduced in Section 2, the process services are responsible for managing session state and preserving process and resource integrity. The control flows inside the executable processes have to be aligned with the wants and needs of business users. Selected business activity services can be instrumented with logging features which can be used for KPI performance management and BAM. Being aware of all service invocations and having access to the service contracts, the ESB can become an integral part of a BAM solution. A key issue here is to avoid that infrastructure elements morph into applications.

Any mature requirements engineering approach adheres to the principle of business alignment; SSE projects this principle to the later phases of the service engineering lifecycle. From an architectural standpoint, *business rules* provide an additional way of expressing business semantics inside an application (architected as an SOA or following another architectural style); such business rules can then be used to express assertions that help to assure that a system meets regulatory compliance laws and other business policies (i.e., manage business integrity). A resulting SSE challenge is how to integrate such business rules into the overall service engineering lifecycle and programming model.

7. Holistic engineering approach. A distinguishing "meta" characteristic of SSE pertains to its holistic nature. More than in traditional software engineering paradigms, SSE demands an interdisciplinary approach towards the analysis and rationalization of business processes, design of supporting software services, their realization, deployment, provisioning, monitoring, and evolution. This implies that SSE concepts, models, and methods are integrated and that SSE tools are interoperable, adhering to open standards and offering integrated support for several stakeholders.

SSE concepts, models/languages, methods, and tools can address/adhere to this tenet by combining and integrating contributions from the fields we mentioned as related work regarding the other six tenets.

4. SSE Research Challenges

To derive research and industry development challenges from the defining tenets and characteristics, a crowd-sourcing and -scoring game was conducted during the SSE seminar at Schloss Dagstuhl. First, the participants were asked to briefly answer the question:

What is the most important challenge of SSE?

32 participants submitted an answer. Next, these answers served as input to a scoring game without any upfront discussion or editing; duplicates were not eliminated. Pairs of answers were scored against each other in four iterations (the pairs were built randomly; the facilitator of the game only was responsible for the time management). The maximum score per iteration was five points. Hence, the highest possible score was 20 points. The result of this sourcing and scoring game is the following consolidated list of answers, ordered by points scored:

1. Address the 'open-world' assumption: unforeseen clients, execution context, usage (16 points)
2. Bridging a modeling chasm: design/develop and delivery/execution (15)
3. 'Open world assumption': uncertainty (15)
4. IT-business alignment, adaptability (15)
5. Alignment of technical and business engineering for services (14)
6. New models and abstractions to represent and handle SOA dynamics (14)
7. To develop software without knowing in which context it is used (14)
8. Programming models and runtime integration (14)
9. Service resilience, system level (robustness) (13)
10. The mapping from requirements to services fulfilling them (13)
11. How to architect SOA with respect to the heterogeneous nature; dealing with heterogeneity (13)
12. Making the leap from business service to the right technical service design (11)
13. Alignment of business and technical SSE level (12)
14. Composability (11)
15. Testing (11)

Not surprisingly, many of these research challenges are closely related to the SSE tenets. Table 1 loosely correlates the 15 research challenges to the SSE tenets. Note that SSE tenet 7 (holistic engineering approach) pertains to all research challenges and is therefore not included in this table.

Table 1. Relating SSE Tenets and Challenges

SSE Tenet	Description	Challenge ID
1	Technical federation	7, 8, 9, 14, 15
2	Dynamism (virtualization)	1, 3, 6, 15
3	Organizational federation	1, 3, 7
4	Explicit boundaries (contracts)	10, 12
5	Heterogeneity	11
6	Business-IT alignment	2, 4, 5, 13, 15

From this informal cross-correlation we may carefully draw first conclusions. It should be noted that the level of granularity of the challenges varies; some are very generic in nature – including challenge 1 and 3 – while others address specific problems such as service composability and testing.

The number of challenges correlated to an SSE tenet indicates how the participants of the game perceive the tenet. The same holds true for the score of the challenge, which is expressed by the challenge ID: a small number indicates high importance.

The research challenges relating to tenet "technical federation" include the design of service-based applications without any knowledge about the context in which these applications will be executed. This research challenge is critical in open and agile service networks, with many interactions between service participants which are not known at design time. In addition, there is a need for novel approaches to integrate programming models and platforms while processes in service networks are executed. The high level of change in service networks also demands services to be robust and reliable. Challenge 8 points out that the traditional boundary between application development and integration on the one hand and application maintenance and change management on the other hand becomes blurred in SSE. In response, *continuous integration*, a term from agile development, may be projected into the operations and maintenance phase of the service development lifecycle to support

continuous evolution. Many backward and forward compatibility issues have to be addressed.

The 'open world assumption' makes the current architecting methods obsolete to a large extent, as they are largely based upon a predefined organizational and technical context. Some flexibility is taken into account, but not nearly as much as required when designing under the 'open world assumption'. Furthermore, the traditional architecture-business cycle [19] that expresses the bidirectional influence between the technical system and the business organization cannot be managed using traditional architecting methods in SSE because of the high dynamism and heterogeneity put forward by the SOA style. Therefore the architecting dimension of SSE needs to be thoroughly re-considered, possibly leading to a new architecting paradigm. Architecture knowledge management with its focus on architectural decisions and their rationale is an emerging sub-discipline of software architecture that we expect to contribute solutions to this new architecting paradigm [20].

Because of the 'open world assumption' and the dynamisms of service-based applications, traditional test methods for system development and deployment are no longer sufficient: as not all usage contexts and configurations can be predetermined in pre-deployment tests setups, tests have to be extended into the operation and maintenance of these applications. Contract-oriented build-in tests, active online tests, and runtime auditors and supervisors are first developments in this direction.

5. Syntheses and Outlook

SOA-enabled applications can be developed and evolved by applying aging software engineering paradigms, notably CBD and OO [21]; however, such conservative approach to SOA design and development leads to certain liabilities that come with these paradigms and, if done naively, a degradation of specific quality attributes such as interoperability, performance (response times, throughout), scalability, and changeability (maintainability). The main reason for these elements of risk is that conventional software engineering paradigms typically adopt a closed world assumption, hypothesizing that applications have clear boundaries, and will be executed in fully controlled, relatively homogeneous, statically assembled, and stable execution environments. This thesis is backed up by conclusions drawn from Boehm's decade-to-decade analysis of software engineering [17].

Instead, we claim that for SOA to be successful, SSE has to embrace the 'open world assumption', in which software services are composed in agile and highly fluid service networks – that are in fact systems of software systems – operating in highly complex, distributed, dynamic, and heterogeneous execution environments. In addition, the service networks that are designed based on this assumption need to be continuously (re-)aligned with business processes, and vice versa. Adoption of the 'open world assumption' is reflected in the seven SSE tenets, which are thus strongly influenced by the underlying distributed computing paradigm and architectural style, SOA.

Based on the research reported in this position paper, we can come up with an initial definition of SSE:

Software service engineering is the science and application of concepts, models, methods, and tools to design, develop (source), deploy, test, provision, and maintain business-aligned and SOA-based software systems in a disciplined, reproducible, and repeatable manner.

Clearly, SSE will benefit from timeless principles and lessons learned from its parent, software engineering. However, we demonstrated that traditional, computing paradigm-specific principles and practices, e.g., CBD, have to be evaluated carefully and, possibly, revised.

In our view, SSE will be based on standards; solution will be frequently realized with Web services of various kinds. Specifications such as SOAP, WSDL, BPEL, WS-Policy, and WS-Agreement already constitute the first step to realize the technical aspects in some of the SSE tenets, including tenets 1, 2, 4, and 5. However, it is evident that research is needed to more effectively satisfy the 'open world assumption'. This has also been reflected in the outcome of the brainstorming session on the key open research challenges.

The research questions that arise from the identified challenges include, but are not limited to:

1. What are the development steps required and patterns eligible when designing and developing service consumers and providers that communicate asynchronously and face a variable QoS mix?
2. How can the vision of service composition, dynamic lookups, and runtime matchmaking be realized without compromising the solutions found to overcome the other challenges, e.g., testing and business alignment?
3. What is the right way of governing SOA design in the absence of a central design authority, in particular the ownership and lifecycle management of services?
4. How to apply design-by-contract within an architectural style which promotes a loosely coupled, platform-independent, and asynchronous integration

model which minimizes the amount of assumptions shared by the communication parties?
5. How to leverage proven OO, CBD, and EAI practices for SSE, for instance agile ones?
6. How to integrate business rules into an SOA and to realize business activity monitoring in the context of the SOA principles and patterns and the SSE tenets?
7. Which related fields can contribute to a holistic and interdisciplinary SSE approach?

These research questions have to be answered as SSE matures; the contribution of this position paper is the SOA-specific selection and refinement of seven engineering tenets and an initial discussion of related research challenges. While none of the tenets in itself is new, their assembly is; as we demonstrated, SOA style-specific issues arise within each tenet. SSE as an emerging discipline must incorporate the tenets as its foundation and meet the related challenges.

The results presented in this article are preliminary in nature. Further work is required in several directions. Firstly, the list of seven tenets has to be validated and possibly refined further. The presented list is derived from a literature survey, and experience from real-world SOA projects, and discussions with leading industry experts and renowned researchers in the field of software engineering, software patterns and SOA. An analysis of more case studies is critical in further validating this initial list. The workshop will serve as a first step to achieve this. Further consolidation of the tenets and research challenges may lead to a future roadmap for SSE.

Acknowledgements

We would like to thank the participants of the Schloss Dagstuhl seminar on SSE [1] for their input and insights which made this paper possible.

References

[1] Software Service Engineering, Schloss Dagstuhl seminar, www.dagstuhl.de/de/programm/kalender/semhp/?semnr=09021

[2] F. Bachmann et al., Technical Concepts of Component-Based Software Engineering, Technical Report, Carnegie-Mellon Univ., CMU/SEI-2000-TR-008 ESC-TR-2000-007, 2nd Edition, May 2000

[3] C. Hofmeister, P. Kruchten, R. Nord, H. Obbink, A. Ran, P. America. A General Model of Software Architecture Design Derived from Five Industrial Approaches. J. Syst. Softw., Elsevier Science Inc., 2007, 80, 106-126

[4] G. Alonso and F. Casati and H. Kuno and V. Machiraju, Web Services: Concepts, Architectures and Applications, Springer, Heidelberg, 2004

[5] B. Meyer, Object-oriented Software Construction, 2nd Edition. Prentice Hall, 2000

[6] G. Hohpe, SOA Patterns – New Insights or Recycled Knowledge? www.eaipatterns.com/docs/SoaPatterns.pdf

[7] M. P. Papazoglou, and W. van den Heuvel. Service-oriented design and development methodology. Int. J. Web Eng. Technol. Vol. 2(4), Jul. 2006

[8] M. P. Papazoglou, W. van den Heuvel, Service oriented architectures: approaches, technologies and research issues. VLDB J. 16(3): 389-415, 2007

[9] C. Szyperski, Component Technology: What, Where, and How? In: Proceedings of the 25th International Conference on Software Engineering International Conference on Software Engineering. IEEE, 684-693, 2003

[10] P. Herzum, O. Sims. Business Component Factory. J. Wiley & Sons Inc., 2000

[11] G. Booch, Object-Oriented Analysis and Design with Applications (2nd Ed.). Benjamin-Cummings Publishing, 1994

[12] I. Jacobson, Object-Oriented Software Engineering. ACM, 1992

[13] R. Popescu-Zeletin, S. Arbanowski, I. Fikouras, G. Gasbarrone, M. Gebler, H. Henning, H. van Kranenburg, H., Portschy, E. Postmann, K. Raatikainen, Service Architectures for the Wireless World. Computer Communications, Vol. 26, No. 1, January 2003, pp. 19 – 25

[14] B. Sarikaya, Principles of Protocol Engineering and Conformance Testing, Ellis Horwood, Series in Computer Communications and Networking, 1993

[15] I. Schieferdecker, J. Grabowski, Advances in Test Automation, STTT Special Issue, Springer Berlin / Heidelberg, ISSN1433-2779, Apr. 2008

[16] B. Meyer, Object-oriented Software Construction (2nd ed.), Prentice-Hall, Inc., Upper Saddle River, NJ, 1997

[17] B. Boehm, A View of 20th and 21st Century Software Engineering. In Proceedings of the 28th International Conference on Software Engineering ICSE, 12-29, ACM Press, 2006

[18] C. Atkinson, D. Brenner, G. Falcone, M. Juhasz, Specifying High-Assurance Services. Computer 41, 8 (Aug. 2008), 64-71

[19] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 2nd Edition, Addison Wesley, 2003

[20] Zimmermann O., Koehler J., Leymann F., Polley R., Schuster N., Managing Architectural Decision Models with Dependency Relations, Integrity Constraints, and Production Rules. Journal of Software and Services, Special Edition on Architectural Decisions, Elsevier, 2009

[21] O. Zimmermann, P. Krogdahl, C. Gee, Elements of Service-Oriented Analysis and Design, IBM developer-Works, 2004