SCA and jABC: Bringing a Service-Oriented Paradigm to Web-Service Construction

Georg Jung[1], Tiziana Margaria[1], Ralf Nagel[2], Wolfgang Schubert[1], Bernhard Steffen[2], and Horst Voigt[1]

[1] Universität Potsdam, Chair Service and Software Engineering
{jung,margaria,schubert,voigt}@cs.uni-potsdam.de
[2] TU Dortmund, Chair Programming Systems
{ralf.nagel,steffen}@cs.tu-dortmund.de

Abstract. Extensibility, flexibility, easy maintainability, and long-term robustness are core requirements for modern, highly distributed information and computation systems. Such systems in turn show a steady increase in complexity. In pursuit of these goals, software engineering has seen a rapid evolution of architectural paradigms aiming towards increasingly modular, hierarchical, and compositional approaches. Object-orientation, component orientation, middleware components, product-lines, and - recently - service orientation.

We compare two approaches towards a service-oriented paradigm, the Service Component Architecture (SCA) and the jABC.

1 Introduction

The Service Component Architecture (SCA) [1,2] was developed recently as an industry standard for service-oriented development of complex, distributed, (web-based) applications.[1] Core of the SCA approach is the notion of the *service component*. Applications are built by arranging a cooperative network of service components which communicate through standardized interfaces.

In essence, SCA is an extensive set of specifications which describe an overall assembly model, implementation support for various programming and database languages, bindings to existing web-service-, messaging-, and middleware-standards, and policy and profiling mechanisms to access and customize infrastructure functionality. As such, SCA has proved effective and useful for building applications in practice.

By continuously emphasizing the service component, service architecture or service oriented development, SCA implicitly promotes a particular concept to be associated with the term "service". A *service* in the sense of SCA is a certain component that provides its functionality through a specific interface (the service interface, often directly identified with the service). Likewise, an *assembly* of services is merely a topology of components which are connected through

[1] The first SCA specification, version 0.9, dates from November 2005, version 1.0 from March 2007.

provide-use relations among their services or service interfaces. In other words, SCA associates service-orientation with a structural, interface-centric view in an assembly model which otherwise follows a component-oriented paradigm (similar to the one proposed in, e.g., [3,4]).

While this notion has its merits in terms of intuition and viability, it is by no means the only workable grounds to introduce a service-oriented paradigm into the practice of application development. An orthogonal view is offered by the service concept of jABC [5,6,7,8,9] a framework for model-driven and service-oriented development that originated in the early '90s (previously named METAFrame [10]). Originally it was applied to the model-driven development of advanced telecommunication services for Intelligent Networks [11,12]. Due to the ease of generalization of that service model, it meanwhile evolved into a flexible model-driven approach spanning both local and distributed (web-based) application development and customization [5,13].

In jABC the term "service" is used to denote functional building blocks (SIBs[2]), which are viewed as independent from their location, the program-entity, and hardware-platform which provides them. Instead, the defining quality of a SIB, which forms the core abstraction of jABC, is its interaction with the environment, which manifests in its behavioural semantics and its manipulations of a global context. The SIBs are assembled – or as one says, orchestrated – with their operational or behavioural semantics in mind. Concretely, this means that each SIB, once activated, executes its logic and upon termination triggers subsequent SIBs according to the outcome of this execution. This methodology of composition has been termed *lightweight process coordination* [9] and is closely related to the SIB model standardized by ITU [14].

The two approaches emphasize dual angles of the idea of a service, which can be characterized as:

– *resource-oriented* vs. *process-oriented*
– *architectural* vs. *behavioral*
– *static* vs. *dynamic*

In the following we will investigate these dual views for their potential to support a truly service-oriented development. We initiate the comparison of the SCA and jABC concepts, considering properties, structures, meta models [15], and semantics, using some examples to illustrate the different viewpoints.

The rest of this paper is organized as follows. Sect. 2 and 3 introduce the meta-models of SCA and jABC, respectively. Sect. 4 examines the common component-middleware paradigm and compares it with the SCA and jABC with respect to structural and computational properties. Sect. 5 discussed the characteristics of component flavored assembly. Sect. 6 evaluates both approaches along technical and pragmatic characteristics. Finally, we briefly discuss related work (Sect. 7), and summarize our findings so far in Sect. 8.

[2] SIB stands for *Service Independent Building Block*, a notion coined in the nineties in the Telecommunication area [14], where the notion of service was meant to denote whole service orchestrations [14,12]. SIBs were back then the atomic service-like entities of which those services were aggregations.

2 The Meta-model of SCA

In SCA, each service component implements some specific business logic and provides its functionality through standardized interfaces, described in SCA literature as service-oriented interfaces and shortly called *services*. To implement its business logic, each service component can rely on third-party functionality provided by other components by linking to their respective service interfaces through so called *references*. A reference interface can be linked to a service interface by means of a *wire*, which abstracts the communication infrastructure through which functionality of third-party service interfaces can be accessed remotely.

A given assembly of service components interconnected through a set of wires can be summarized as a *composite*, and service or reference interfaces of the components inside the composite can be *propagated* to be visible as interfaces of the composite itself. Thus, the composite as a whole shows similar characteristics as a single service component and can be used in the same way inside larger composites. This enables a straightforward hierarchy structure for SCA assemblies, distinguishing "atomic" components and composite components. Finally, a service component may feature property-interfaces which allow to customize its behavior and can be propagated to be visible as properties of a composite in the same way as interfaces.

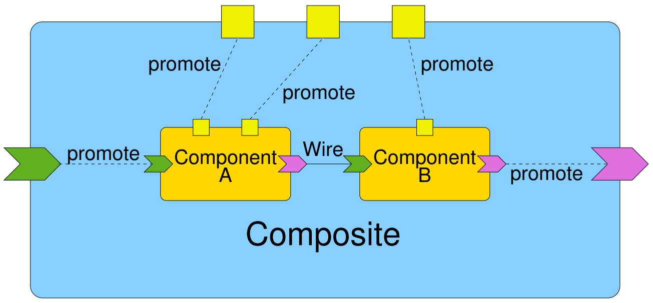

Fig. 1. Schematic of the SCA assembly-model

Fig. 1 shows a schematic of the SCA assembly model (see `www.osoa.org`). Other than what one might expect, the service interfaces are the incoming interfaces and the reference interfaces are the outgoing ones. This reflects a view from *inside* the component/composite, where services (of others) are used and the references to services (of oneself) are provided. It can therefore be described as a developer's view (as opposed to, e.g., a composer's view), which again emphasizes the idea that services are used to integrate third-party functionality into an application.

In [2], Edwards proposes a UML-model of the SCA assembly. The excerpt in Fig. 2 shows the essential part: concrete components with their component-properties, -references, and -services serve as implementation for component

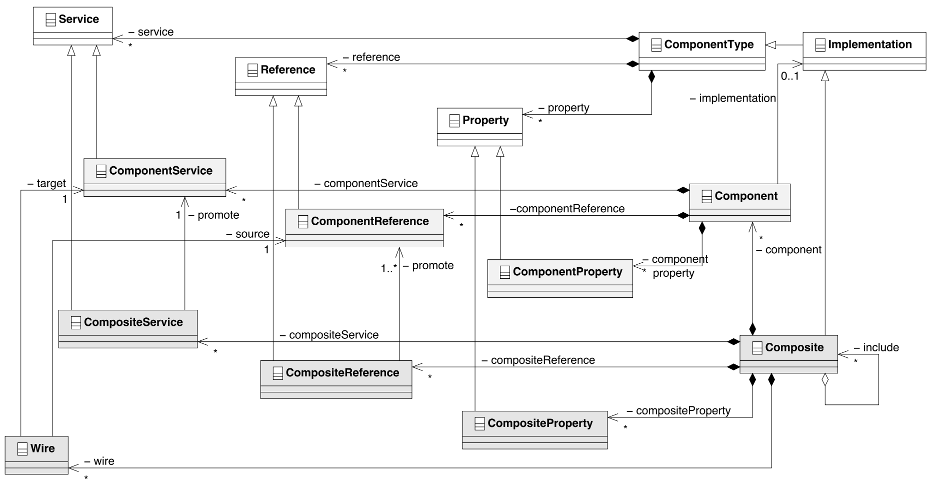

Fig. 2. An UML-model of the SCA-assembly

types. The composite on the other hand is a specific way to implement a component type (indicated by the inheritance relation to implementation), and can in turn contain multiple components. The wire as a part of the composite connects one reference with one service interface. Unfortunately, this model leaves out several crucial interrelation constraints, for example it shows that one service and one reference can be attached to one wire, but it glosses over how many wires can be attached to a single service or reference. E.g., a single wire per service interface and zero-to-many wires per reference interface seem reasonable connectivity constraints. Also, the model shows inheritance relations across different levels of abstraction (i.e., type–instance relations. Services on components *inherit* from services on component types in this model. Here an implementation/interpretation relation would seem more appropriate. Nevertheless, the model is clearly meant to expose *structural* or *static* interrelations of the SCA notions to enable implementations. To this aim it is certainly more helpful than a conceptional relationship model.

3 The Meta-model of jABC

jABC follows a completely different compositional paradigm, called lightweight process coordination [9], which - instead of structural properties - revolves around the operational aspects of its elementary building blocks (see Sect. 1). Concretely, a SIB is an executable entity, internally realized as a specifically annotated Java class.[3] As such, it intrinsically carries an arbitrarily fine-grained/precise operational semantics. A SIB can be a model placeholder for some functionality

[3] While jABC is currently implemented in Java, the core concept is independent of the programming language. Previous versions, for example, realized the same model in C++.

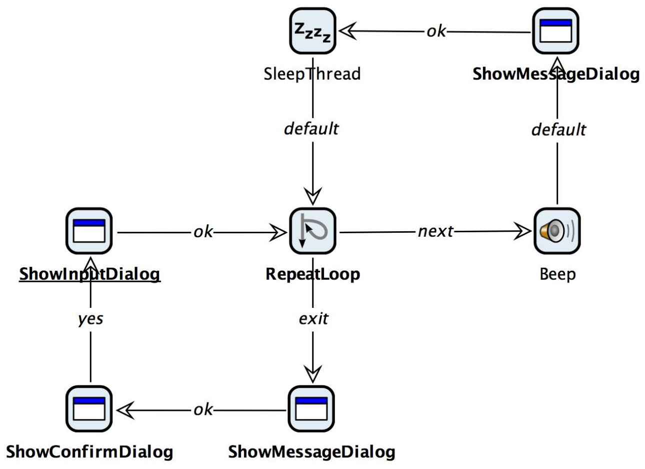

Fig. 3. Assembly of services in jABC: the SLG is a *process*

or a full implementation of that functionality, as well as any level of refinement/abstraction expressible by the (Java) programming language in between. Further, each SIB has one entry point, where the execution starts, and multiple exit points (called *branches*) which represent different outcomes of its execution at the model level.

SIBs can be arranged into topologies called *Service Logic Graphs* (SLG) which specify process behavior by connecting outgoing SIB branches to the entry points of other SIBs. Inside an SLG, the execution of a SIB starts whenever one of its incoming branches is *active*, which means that the SIB which governs the branch terminated its execution with an outcome associated with that branch. One SIBs inside an SLG can be assigned to be *start SIBs*, which means that its execution is started without an incoming active branch; start SIBs are the entry points of the process modelled by the respective SLG.

Fig. 3 shows a simple process, graphically modeled as SLG.[4] The labels of start SIBs, here **ShowInputDialog**, are underlined, those of possible exit SIBs (e.g., **ShowInputDialog**, **RepeatLoop**, ...) are printed in bold-font. In the basic, sequential case, each SIB terminates with one active branch which determines the next SIB to be executed. Parallel and concurrent structures are likewise possible, as used in the bioinformatics applications [16,17]. Hence, an SLG is a graphical, executable, node-action process description.

SLGs can be canonically wrapped into (*graph-*) SIBs to allow for a hierarchical organization of complex process models. Moreover, process models which follow a certain standard defined by jABC can be directly exported into (partial or complete) stand-alone applications, a feature which turns jABC from a modeling into a development tool. Finally, there are SIBs which serve as wrappers for outside

[4] The process model shown in Fig. 3 is one of the tutorial examples which come with the standard installation of jABC.

functionality (e.g., non-Java applications such as C++, C#, SOAP/WSDL Web services, etc.): this enables modeling and building of heterogeneous, distributed, applications.

The service concept as a compositional paradigm is particularly strong in jABC, since all visible business-logic in an SLG boils down to *orchestration* of the functionality abstracted within the SIBs. Each SIB independently and without interruption manipulates the global context, similar to what happens in blackboard systems [18], and upon its termination the jABC passes the control to the next SIB. As opposed to a component-oriented approach, SIBs never access or interact with other SIBs through channels or interfaces; instead, their functionality is local and self-contained.

4 Comparison with Component-Orientation

If one revisits the SCA meta-model with a regular, component oriented, architecture-methodology in mind (such as, e.g., the CORBA Component Model CCM [19], or Enterprise Java Beans [20]), many structural similarities surface. All of them support the concept of the *component*, accompanied by the notions of the *interface* which offers access to the component, and the *connector*, which allows composers to link components together. The variety of kinds of available connectors, components, and interfaces is sometimes associated with the term *architectural style*, depending on the communication capabilities it offers (e.g., publish-subscribe architecture, remote procedure-call RPC, broadcast, etc.).

These three fundamental concepts (colloquially: boxes, dots, and lines) generally appear in a middleware context or comparable setting, where a more or less fixed infrastructure with a given set of communication, persistence, execution, and similar capabilities is abstracted to be able to focus on business-level functionality and high-level assembly. Fig. 4 for example formalizes EJB in a two-part meta-model which distinguishes between general parts of a component-oriented paradigm (labeled platform independent model, PIM) and parts specific to the EJB model (labeled platform specific model, PSM).[5] Meta-models of various middleware-centric component frameworks can be built by simply exchanging the PSM with the specifics of a different platform.

SCA fits into this structural model too, if one considers the use of (previously existing, established) internet-communication protocols (HTTP, SMTP, etc.) to be comparable to more local approaches such as a CORBA RPC layer. In this case, a meta-model for SCA can be built by using the aforementioned PIM and complementing platform-specific parts and terms of SCA (Fig. 5).

Note that in middleware-centric architectural styles, the term "service" is often used for the capabilities of the middleware (e.g., persistence service, publish-subscribe service, etc.). These middleware services are again fundamentally distinct from the service concept of SCA. In high-level, and in particular business process level service oriented environments, middleware services have to

[5] The notions of PIM/PSM have been proposed in similar forms independently by various authors; see, e.g., [15], where the acronyms PIM and PSM appear first.

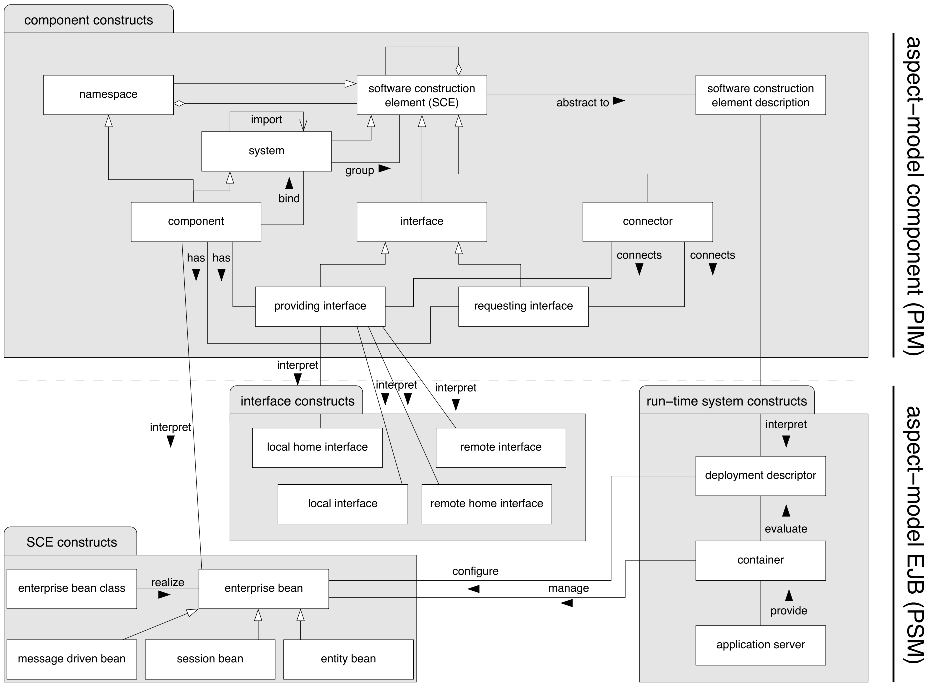

Fig. 4. A PIM/PSM model of the EJB component-middleware platform

be thoroughly abstracted so that they can be independently and mechanically configurable. This is because, to actually facilitate development instead of making it more complex, they have to support an agnostic developer (i.e., component integrator). For example in a CCM architecture [19], no matter where a component is located, it can use the RPC service, hence the service has to be location-agnostic. The middleware service is therefore not analogue to the service (interface) in SCA which is location-bound. Instead the wire in SCA is a much closer equivalent to the location-agnostic intuition of a service because it abstracts a ubiquitous communication service. In fact, keeping the complexity of the infrastructure and communication channels abstracted in SCA in mind, it seems reasonable to consider their entirety as some kind of middleware.

The assembly model of jABC follows an entirely different approach which is operation-centered, instead of structure-centered. As opposed to the SCA service interface it does also not consider location. It does not correspond in any way to the component-oriented, middleware-centric, paradigm. In fact, the notion of the global context in jABC contradicts the strict data-encapsulation which is required by the component methodology. There is, however, a close correspondence between the idea of the SIB and the (arbitrarily complex) middleware-service. Like the SIB, the middleware service does not consider a localized persistence feature in its definition, and like the SIB, it is defined rather through its functionality than its structure.

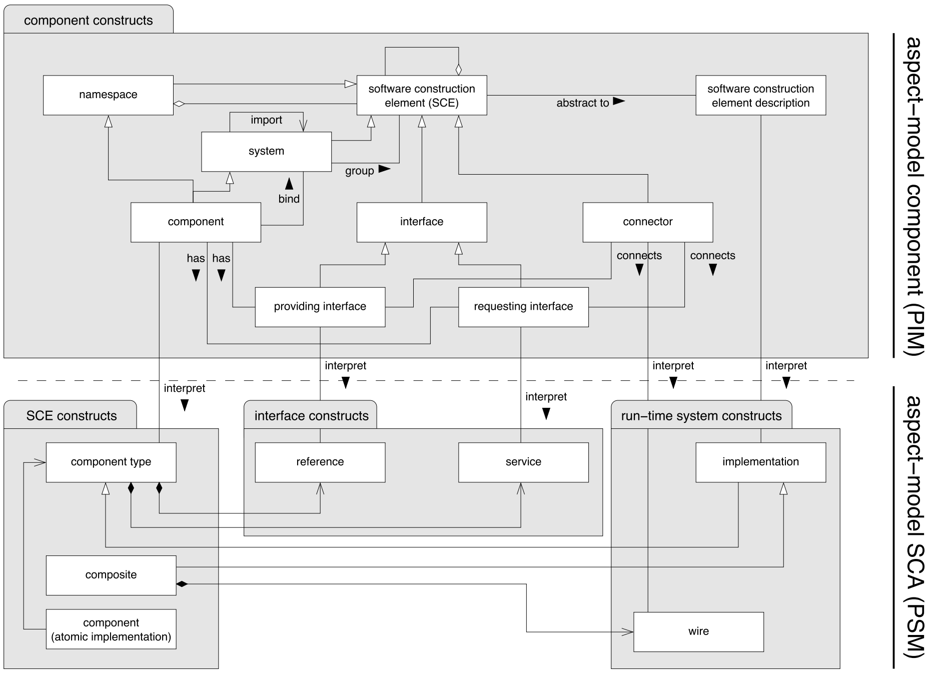

Fig. 5. SCA meta model, fitted into a component-middleware structure, cfr. Fig. 4

5 Characteristics of Component Flavored Assembly

We discuss here the two aspects that seem to us most prominent.

5.1 Complex, Fixed, Layer-Structures and the Service Concept

The two notions of service discussed in the previous section (the middleware/ infrastructure-layer service and the SCA service interface) fall short for the task of service composition. In both concepts (as opposed to the one of jABC) the service is only modeled through the structural properties of its access point, and if services are to be combined the developer has to resort to hand-coded business logic.

At the same time, even the complexity of the infrastructure layer itself suggests the necessity for a methodology for easy assembly of services. The fact that for example connectors in a component topology cannot be neglected as trivial was first pointed out by Shaw [21], and this realization subsequently found its way into literature about practical application of the component-oriented paradigm (e.g., in [3, *pp.* 429] Szypersky states that “A connector, when zooming in, can easily have substantial complexity and really ask for partitioning into components itself”). Nevertheless, discovering this “duality” between component and connector [3, *same page*] did not yet trigger a revisiting of terminology,

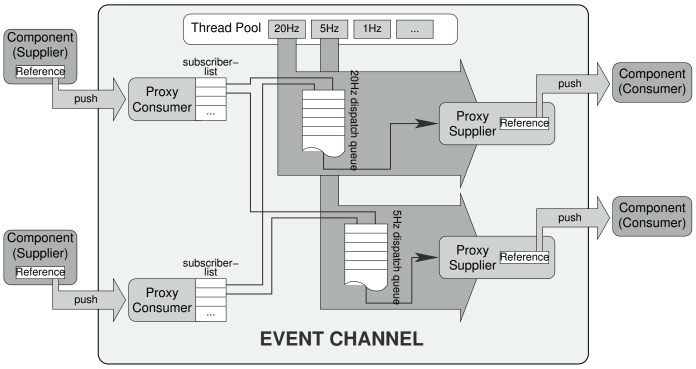

Fig. 6. Schematic view of the PRiSM Event-channel

paradigms, and abstractions of component-oriented architectural styles or an introduction of lightweight process coordination into the middleware concepts.

To illustrate the lack of expressiveness, consider the PRiSM real-time component middleware, which was developed by the Boeing company within the Bold Stroke effort for middleware-based aviation control systems [22]. PRiSM features, much like CCM, two main channels for communication: An asynchronous event notification service with very limited payload capability and a synchronous RPC connection which can be used to communicate data through the return values (but not to trigger computation, since it is not thread-safe). The implementation of the notification service called *event-channel* is intertwined with the middleware's thread handling mechanism (Fig. 6). A thread starts with a periodic timeout event (20Hz, 5Hz, 1Hz in Fig. 6, distributed through the event notification service) and runs until all events within its buffer (called the rate group's *dispatch queue*) are handled. The events trigger computation within the individual components and subsequent events can be queued within the same dispatch queue or dispatch queues of other threads. In other words, timeout events and dispatches drive the components of a rate group, and the buffering within the event channel serves for messages to cross rate groups.

To exchange data between different threads/rate-groups, systems on the PRiSM platform make heavy use of a so-called *control-push–data-pull* strategy. If new data has been generated (e.g., by a device-driver component such as a GPS), the generating component issues an event which notifies consumers of the data. This event is queued within the dispatch queues of the receiver's respective threads. At the time these threads become active and the event is dispatched, the receiving components actively fetch the announced data from the generating component through RPC-calls. This method guarantees that threads never execute out of turn, and data is only transmitted when both available and needed. The drawback is that two different middleware connectors are needed for one logical connection.

To avoid "cross-wiring" or other inconsistencies which can occur with these double connections it seems obvious that a new abstraction should be added

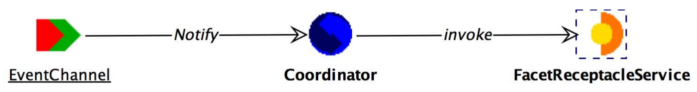

Fig. 7. Composing a non-blocking message service in jABC

to the middleware capabilities which denotes the double connection as a single connector of a new type. In a middleware-centric or generally in a tiered approach with emphasis on interface compatibility, we can certainly introduce the abstraction, but the semantics of the new connector can only be implemented "by hand", since the operational semantics of the different existing connectors are not captured by the model, even though the general push-pull strategy could have been understood without knowledge about the implementation details of the individual communication channels.

When using a lightweight process coordination methodology, as in the jABC, instead (applied to coordinating the middleware processes), the task of assembling a new (ubiquitous, semantically unambiguous, hierarchically constructed) communication service from existing ones becomes clear and simple. The two connectors are composed as a sequence, together with a control unit which handles possible data conversions if necessary (Fig. 7).

In the PRiSM middleware, the infrastructure services are necessarily simplistic due to the real-time aspects, with limited options to reasonably combine multiple services into orchestrated compounds. Yet, even in this constrained microcosm-setting the middleware would benefit from process coordination. In the macrocosm of a web-based, highly distributed, application, the variety of existing services increases: there are services which would be considered infrastructure (such as communication), and services provided on top of the infrastructure. Here the possibilities of assembling meaningful combinations multiply.

5.2 Perspective, Location, and Entry-Point: Topology vs. Coordination

Consider a composite service where a central unit C acts as the orchestrator and in turn relies on services provided by distinct units A and B. For example, for a list of authorized database-accesses, A authorizes the access, B offers it, and C orchestrates services A and B to allow its users to handle the authorization and the complete list of accesses via a single call to C.

The difference between a model of this situation in SCA and in jABC can be summarized as *perspective* or viewpoint (Fig. 8 (a) and (b)).

- SCA offers a model of the physical *topology*: C appears as the provider of the service that promotes a reference for service access. Through wires, C connects the service interfaces it needs to the respective references provided by A and B. The structural aspects of the example are captured and localized by the SCA model, but the operational aspects are hidden. In fact, SCA would require to integrate tailored business logic into C as their orchestration.

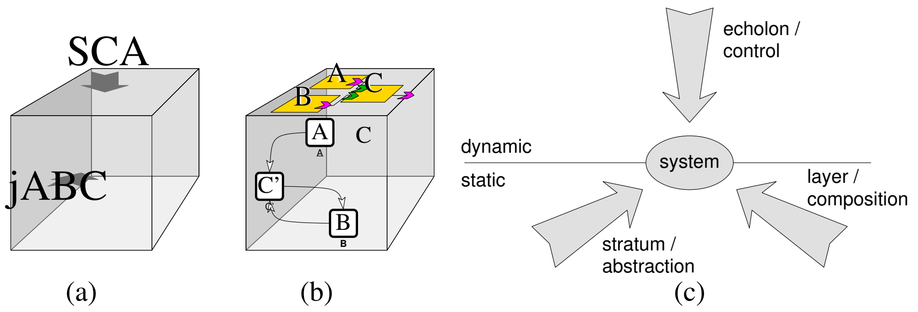

Fig. 8. Different perspectives: (a) SCA and jABC perspectives, (b) Topology or process coordination, (c) Perspectives according to Mesarovic

– jABC on the other hand abstracts all location aspects. Entry point of the operational model *through* the composite-service C is the start-SIB A, where the first operation happens (the authorization), then C' (the actual mediation service) is invoked, and subsequently a loop between C' and B performs the service until the list of authorized accesses is complete. The jABC model glosses over the actual topology of the involved units (components, or SIBs, or services).

Concerning executability and coding aspects, while the SCA-model only allows auto-generating code-stubs which handle the interconnection and communication aspects, but needs control and business logic to be added, the jABC model is designed to be sufficient to generate the complete system through the Genesys plugin [23]. As described in Sect. 3, given a sufficiently complete modeling of the individual SIBs, jABC acts as an intuitive, graphical, development tool rather than a modeling tool.

The need for multiple perspectives has been noted earlier. Most eminently in [24], Mesarovic et al. propose three perspectives which have to be consolidated for a complete system model. They coin the terms *echelon*, *layer*, and *stratum* which—in current terminology—denote control or behavior, composition or topology, and abstractions and data types respectively (Fig. 8(c)).

6 Evaluation

In this section we are going to investigate the appropriateness of the described specification styles for *service orientation* relative to the widely agreed upon characteristics of service orientation. Here, we consider a technical and a pragmatic side, both with three dimensions.

Technical characterization: as introduced in [7]

– (Extreme) loose coupling and self containment of the services.
– Virtualization: clear separation from implementation/realization details.
– Domain specificity: a service-oriented setup should seamlessly integrate into the setup of the considered domain.

Pragmatic characterization:

– Scalability, both for the successive 'assembly' of functionality, as well as for the number of users of developed artifacts.
– Participation: service-orientation aims at giving the domain expert access and control of the development and evolution of the artifacts.
– Agility: changes and adaptations should be easy and ideally be controllable at the (process) model level.

Despite their strong semantic differences, the SCA approach and jABC approach are quite similar when it comes to the technical characteristics: both support loose coupling and virtualization, and the organization in the virtualized components enables a domain-specific development.

The differences show up, however, when it comes to the pragmatic characteristics, which we will now consider individually.

6.1 Scalability

This first dimension is still supported quite similarly in the two approaches, by clean concepts of hierarchy. Both SCA and jABC offer conceptually similar options to assemble larger elements (service components or SIBs) out of topologies of smaller ones: In SCA the composite can act as component, in jABC an SLG can be packed into a SIB, and in either case it is necessary to propagate or mark interfaces of the internal structures to be visible on the external structure.

Concerning the scalability in the number of users, both approaches can make adequate use of standard technology like scalable application servers, which support growing sizes of users.

The real difference between the SCA and the jABC approach becomes apparent when looking at the remaining two dimensions.

6.2 Participation

Technically, we can regard service orientation as an 80/20 approach to application/process development. It aims at a maximal involvement of the application expert in order to avoid misunderstandings and to overcome communication hurdles. At the best, users should be able to directly influence, control, and adapt the services according to their needs. At least for typical day-to-day situations, this should be possible without IT knowledge. Thus, service orientation potentially has a disruptive impact on the current structures.

The jABC directly addresses this goal by putting the application/business process in the center of attention, while the architectural and resource-oriented SCA approach still addresses IT experts.

6.3 Agility

Agility can be regarded as a logical consequence of rigorous participation. Giving control (of the 80%) of system adaptation and evolution directly to the application expert eliminates time consuming and expensive multi-party interactions, with the misunderstandings and communication hurdles for the majority of tasks.

The One-Thing Approach supported by the jABC [25,26] is directly designed to establish this level of control: the user/application/business-level process remains part of the artifact, which gradually turns into the product along the development and which is maintained during the subsequent lifecycle. This allows the application experts in particular to redesign their processes, control permissions, and add business rules at the application/business process level, with the immediate consequence of enactment. Thus, essentially, the changes are implemented as soon as they were specified. Of course, more radical changes will still require IT support, but in our experience they are not as frequent. As before, the SCA approach can be seen here as a valid support for IT involvement. Thus it may well accelerate required modifications, but in a more 'classical' setting.

7 Related Work

Previous work in service oriented architecture research focusses mostly on standards, languages, and features of SCA [27,28], or on the assembly model (i.e., interface definitions) [29]. There is little work on classifying the architectural patterns or combining them with flexible behavioral semantics.

Among the approaches towards combining service orientation with general-purpose behavioral descriptions is the SENSORIA project [30] and [31,32]. SENSORIA aims at a comprehensive approach to service-oriented development with focus on specific problems of loose coupling and heterogeneous environments, raising issues in security, specification, and communication, at a technical level. In contrast, we focus on participation, meaning that we directly address and involve the application expert via the 'One-Thing Approach' [33,25], throughout the entire lifecycle.

8 Conclusions

Both SCA and jABC are frameworks with substantial practical merit. By emphasizing the term "service" within the basic modeling structures, they both also claim to move forward to a novel, service-oriented, software-development paradigm. Nevertheless, their notion of service is fundamentally different.

This paper presented a structural, concept oriented, comparison between these two approaches, focussing on the main characteristics and of service orientation. We showed that

– The SCA development paradigm is essentially *component-oriented*, and as such it treats its extensive infrastructure specification as analogous to a middleware layer. Therefore it builds on proven software construction methodologies which are established as best practice in industrial software development, and brings them into the realm of web-based application development.

- By elevating the required interface, called *service*, to be the core modeling entity, SCA deviates from the standard component-oriented paradigm, which instead puts the component itself into the center of consideration. It seems however questionable whether this shift of emphasis alone is sufficient to warrant the label "service-oriented" development.
- As common to other component-oriented approaches, the operational aspects of a software system are not captured within SCA models, which concentrate on their structural aspects. This could become a handicap when addressing problems as service orchestrations, where SCA can rely on strong capabilities of the comprehensive infrastructure (i.e., a vast body of specification and machinery, XML artifacts and ties to all major communication protocols, maintained by a large community), but still needs hand-tailored solutions to be supplied for control-flow.
- The jABC methodology on the other hand is entirely operation centered and it hides topology, location, and connection aspects. It appears as the better candidate when it comes to transcending the semantic gap, as even control structures exist as services. While the ties to web-communication protocols are not an essential part of jABC, they are provided through various plugins (most eminently through jETI).
- The service concept of jABC is very close to an intuitive understanding of service (which, e.g., manifests itself in the term "middleware service" and in various other domains) that requires the service to be ubiquitously accessible (location-agnostic) and mechanically configurable. In fact it seems that the lightweight process coordination offers an elegant way to recombine and enhance common platform services as well as complex web-based business services. Therefore, jABC is not only applicable to web-development or similar tasks, it also offers itself as the semantic underpinning for an operational modelling inside component-oriented methodologies.

Looking at the intent and main characteristics of service orientation, it became clear to us that the two dual specification approaches, although being semantically quite different, are quite similar concerning the first four criteria, namely loose coupling, virtualization, domain specificity, and scalability. In fact, both approaches are based here almost on the same means – only applied to frameworks that aim at covering different perspectives: SCA takes the architectural perspective, which focusses on a resource view, and jABC a behavioral perspective, which focusses on a process view.

The real impact of the choice of perspective, however, becomes apparent when looking at the remaining characteristics, namely participation and agility, and this essentially for one single reason: Whereas SCA is based on lower level, infrastructure-oriented modelling and design, which is accessible to typical domain experts, jABC puts the (user-level) process in the center of attention. This directly supports participation, and, due to the One-Thing Approach, it also provides a new level of agility: the majority of day-to-day change requests can be resolved directly at the application process level, without even involving IT support. Put figuratively, the jABC is a framework that support the slogan "Easy for the many, difficult for the few", in particular by enabling the *many*, whereas SCA addresses the *few*, and supports them in their role of solving difficult tasks.

References

1. Margolis, B., Sharpe, J.L.: SOA for the Business Developer. MC Press (June 2007)
2. The Open SOA Collaboration: SCA web-site, `http://www.osoa.org/display/Main/Service+Component+Architecture+Home`
3. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. ACM Press / Addison-Wesley (2002)
4. Heineman, G., Councill, B.: Component-Based Software Engineering: Putting the Pieces Together. Addison Wesley, Reading (2001)
5. Steffen, B., Margaria, T., Nagel, R., Jörges, S., Kubczak, C.: Model-Driven Development with the jABC. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 92–108. Springer, Heidelberg (2007)
6. Homepage of the jABC framework, `http://www.jabc.de`
7. Margaria, T., Steffen, B.: Service engineering: Linking business and IT. IEEE Computer 39(10), 45–55 (2006)
8. Steffen, B., Narayan, P.: Full life-cycle support for end-to-end processes. IEEE Computer 40(11), 64–73 (2007)
9. Margaria, T., Steffen, B.: Lightweight coarse-grained coordination: a scalable system-level approach. STTT - Int. Journ. on Software Tools for Technology Transfer 5(2), 107–123 (2004)
10. Steffen, B., Margaria, T.: METAFrame in practice: Design of Intelligent Network Services. In: Olderog, E.-R., Steffen, B. (eds.) Correct System Design. LNCS, vol. 1710, pp. 390–415. Springer, Heidelberg (1999)
11. Steffen, B., Margaria, T., Claßen, A., Braun, V., Reitenspieß, M.: An environment for the creation of intelligent network services. In: Annual Review of Communication, Int. Engineering Consortium (IEC), Chicago, USA, pp. 919–935 (November 1996)
12. Margaria, T., Steffen, B., Reitenspieß, M.: Service-oriented design: The roots. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 450–464. Springer, Heidelberg (2005)
13. Kubczak, C., Margaria, T., Steffen, B., Nagel, R.: Service-oriented Mediation with jABC/jETI. In: Petrie, C., Lausen, H., Zaremba, M., Margaria, T. (eds.) Semantic Web Services Challenge: Results from the First Year (Semantic Web and Beyond). Springer, Heidelberg (to appear, 2008)
14. ITU Geneva, Switzerland: Recommendation Q.1211 - General Recommendations on Telephone Switching and Signaling Intelligent Network: Introduction to Intelligent Network Capability Set 1 (March 1993)
15. Object Management Group: MDA guide version 1.0.1, `http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf`
16. Lamprecht, A.L., Margaria, T., Steffen, B.: Seven variations of an alignment workflow – an illustration of agile process design/management in Bio-jETI. In: Măndoiu, I., Sunderraman, R., Zelikovsky, A. (eds.) ISBRA 2008. LNCS (LNBI), vol. 4983, pp. 445–456. Springer, Heidelberg (2008)

17. Lamprecht, A.L., Margaria, T., Steffen, B., Sczyrba, A., Hartmeier, S., Giegerich, R.: Genefisher-p: Variations of genefisher as processes in biojeti. BioMed Central (BMC) Bioinformatics 2008. In: Supplement dedicated to Network Tools and Applications in Biology 2007 Workshop (NETTAB 2007), April 25, vol. 9(Suppl. 4), p. 13 (2008)
18. Nii, H.: Blackboard systems. AI Magazine 7(2), 38–53, 7(3), 82–106 (1986)
19. Object Management Group: OMG formal/06-04-01 (CORBA Component Model Specification, v4.0) (April 2006)
20. Matena, V., Krishnan, S., DeMichiel, L., Stearns, B.: Applying Enterprise JavaBeans. Addison Wesley, Reading (2003)
21. Shaw, M.: Procedure calls are the assembly language of software interconnection: Connectors deserve first-class status. In: Lamb, D.A. (ed.) Selected papers from the Workshop on Studies of Software Design. LNCS, vol. 1078, pp. 17–32. Springer, Heidelberg (1993)
22. Hatcliff, J., Deng, W., Dwyer, M., Jung, G., Ranganath, V.P.: Cadena: An integrated development, analysis, and verification environment for component-based systems. In: Proc. 25th Int. Conf. on Software Engineering (ICSE 2003), May 2003, vol. 841, pp. 160–173. IEEE Computer Soceity Press, Los Alamitos (2003)
23. Jörges, S., Margaria, T., Steffen, B.: Genesys: Service-oriented construction of certified code generators. ISSE – Int. Journal on Innovations in Systems and Software Engineering – a NASA Journal (to appear)
24. Mesarovic, M., Macko, D., Takahara, Y.: Theory of Hierarchical, Multilevel, Systems. Mathematics in Science and Engineering, vol. 68. Academic Press, New York (1970)
25. Margaria, T., Steffen, B.: Business Process Modelling in the jABC: The One-Thing Approach. In: Handbook of Research on Business Process Modeling, IGI Global (2008)
26. Margaria, T.: Service is in the eyes of the beholder. IEEE Computer 40(11), 33–37 (2007)
27. Curbera, F.: Component contracts in service-oriented architectures. IEEE Computer 40(11), 74–80 (2007)
28. Zou, Z., Duan, Z.: Building business processes or assembling service components: Reuse services with bpel4ws and sca. In: ECOWS 2006, Proc. European Conference on Web Services, pp. 138–147 (2006)
29. Ding, Z., Chen, Z., Liu, J.: A rigorous model of service component architecture. ENTCS 207, 33–48 (2008)
30. Wirsing, M., Hölzl, M., Acciai, L., Banti, F., et al.: SENSORIA patterns: Augmenting service engineering with formal analysis, transformation and dynamicity. In: ISoLA 2008. CCIS, vol. 17, pp. 170–190. Springer, Heidelberg (This Volume) (2008)
31. Fiadeiro, J.L., Lopes, A., Bocchi, L.: Algebraic Semantics of Service Component Modules. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) WADT 2006. LNCS, vol. 4409, pp. 37–55. Springer, Heidelberg (2007)
32. Fiadeiro, J.L., Lopes, A., Bocchi, L.: A Formal Approach to Service Component Architecture. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 193–213. Springer, Heidelberg (2006)
33. Hörmann, M., Margaria, T., Mender, T., Nagel, R., Steffen, B., Trinh, H.: The jabc approach to rigorous collaborative development of scm applications. In: ISoLA 2008. CCIS, vol. 17, pp. 724–737. Springer, Heidelberg (This Volume) (2008)