Applying Service-Orientation Through a Reference Architecture

Helge Hofmeister
Distributed and Mobile Systems Group, Otto-Friedrich Universität
Bamberg, Germany
Email: hofmeisterh@gmail.com

and

Guido Wirtz
Distributed and Mobile Systems Group, Otto-Friedrich Universität
Bamberg, Germany
Email: guido.wirtz@uni-bamberg.de

ABSTRACT

This article investigates the application of the service-oriented architectural style in the context of large organizations. It introduces an architectural reference framework that allows for a business process-centered development of composite applications. The framework groups artifacts of similar abstraction levels as well as concerns at five distinguished layers. This way, the service-oriented principles of *abstraction* and *autonomy* can be respected when designing applications. The layers also correspond to phases of a design methodology and cover the aspects of composite applications from process-centered orchestration, over transactional coordination to data transformation and connectivity. Based on the framework this article shows as well how an integration design methodology can be used to leverage the application systems in the context of a given business process.

Keywords: Service-Oriented Architectures, Enterprise Application Integration, Reference Architecture, Design Methodology

1. Introduction

Service-oriented architecture (SOA) or service orientation (SO) is an architectural style that allows the construction of applications that reuse distributed functionality of heterogeneous application landscapes. Applications that reuse functionality and expose their functionality as web-based applications are so-called composite applications (e.g. [1]). Together, SO and composite applications promise to protect investments in legacy landscapes by reusing the existent functionality while allowing for the incorporation of recent business changes.
Garlan identified uncertainty about the control model as a major issue when building systems that reuse existing parts (cf. [2]). This issue persists when building service-oriented composite applications.
One way to address this issue is the concept of Business Process Integration Oriented Application Integration (BPIOAI) introduced by Linthicum (cf. [3]). This concept centralizes the control model outside the participating application systems and uses business processes as the central control instance over distributed functionality. This functionality can be exposed by the means of services that have a formally described interface (cf. [3]).

Papazouglou stated that an SOA allows business process-centered control over distributed services by introducing process-centered service aggregation, or so-called service orchestration. The latter is introduced as a part of a service-oriented architecture. It serves as a mechanism for aggregating basic services to more specialized services (cf. [4]).

From the proposed aggregation of services, another possible benefit for the industry can be identified: required changes for functional enhancement can be realized as additional services that are aggregated together with services that expose standard functionality of "Commercial off-the-shelf software" (COTS). Such aggregators could thus provide the required functionality which is typically offered by separate systems. This way SO could also contribute to keeping COTS unmodified – which is a major aspect of today's IT governance (cf. [5, pp. 69f.]).

Schelp and Winter state that there are only few approaches to structure composite applications such that service-oriented principles are incorporated while actual requirements are realized in a heterogeneous application landscape (cf [6, p. 68]).

This article aims at closing this gap. It introduces an architectural framework that provides a meta-structure for composite applications. As such, it allows organizations to develop composite applications in a standardized way while using actual business processes as the input for the design of a application. Special attention is paid to include mechanisms for integrating existing applications of an organization's system landscape.

After presenting related work in section 2, we introduce our architectural framework in section 3. The introduction is followed by the discussion of a methodology that can be used to apply the reference architecture to an actual problem (section 4). We close with a summary and an outlook to future work in section 5.

2. Related Work

The Business Process Integration Oriented Application Integration (BPIOAI) approach introduced by Linthicum in [3] is one key concept for the findings of this thesis.
Without explicitly referencing the work of Linthicum, Hentrich and Zdun use BPIOAI to put a service composition layer on top of a service oriented architecture (cf. [7] and [8]). This service-oriented reference architecture emphasizes the distributed nature of SOA by incorporating service invocation, adaptation, request handling and communication into the framework. The service coordination is classified as a macro workflow for business processes and a micro workflow for so-called "more technical" aspects. This way, a business process-centric development of composite application can be achieved.
Erl has established in [9] a reference architecture that distinguishes a service interface layer, an orchestration layer, a business service layer and an application service layer. The service interface layer is put as a mediator in between the business process layer and the application layer.
The outline that is given by [10] also layers business processes, orchestrated services and enterprise components on top of an ap-

plication landscape. While stating that user interfaces are out of scope for the discussions around a SOA, the reference architecture of [10] anticipates that a dedicated user interface layer might be needed in the future. It is stated that, however, such a layer will be placed on-top of a business process layer and access services offered by this layer or by the basic service layer.

The ASG project [11] aims to increase an organization's flexibility by defining a platform for automated service composition and enactment. This objective is addressed by introducing semantic annotations in addition to the syntactic definition of services. Such approaches are considered the next step after initially introducing SO in large organizations.

3. Architectural Framework for Composite Applications

According to Linthicum, Business Process Integration Oriented Application Integration (BPIOAI) provides another layer on-top of existent system integration such as Information Oriented Application Integration (IOAI) or Service-Oriented Application Integration (SOAI) (cf. [3]). This means, that system integration-focused technologies such as eg. JMS messaging (for IOAI) or HTTP-based web services (for SOAI) are controlled by a top-level orchestration layer that implements the business logic. Often, to this business-process implementation, it is referred to as composite application. While the composite application implements the business logic, the used integration technologies solely provide the abilities of calling application systems that in turn provide the logic that is orchestrated by the composite application. To this part, not implementing any business logic, it is referred to as integration sub-system.
This section presents five layers that together form the complete composite application including the integration sub-system.

Reference architectures for service-oriented applications usually incorporate abstraction by the notion of several *layers*. The reference architecture introduced here also incorporates this principle. It uses a central service orchestration layer that aggregates services exposed by less abstract layers. The reference architecture allows for a business process-based definition of this orchestration. For this sake it abstracts from "technical" details. It includes a data repository for context handling that also participates in the management of distributed transactions. This, of course, increases the extent of coupling between the central elements of the composite application (cf. [5, pp. 185ff]). This is necessary to create efficient composites that can be deployed on arbitrary platforms (cf. [5, pp. 191ff]).
The notion of an eventing system is used for consistency management within the context and for de-coupling the different processes and tasks that are possibly supported by one composite application.
By connecting these components, the reference architecture defines a second layer of service aggregation — the *service coordination layer*. On this layer, if required, orchestrated entities from the top layer can be described as aggregations of functionality that is exposed by systems of the application landscape. It provides a mechanism for the *mediation* of the services. This coordination includes the "details" of transactional management and connects application systems either directly to the composite using the common service protocol or by using the *data exchange and data transformation layer*.
This layer ensures, together with the layer of *connectivity*, various ways of interactions with back-end systems while abstracting from communication semantics.

This reference architecture allows for defining service boundaries in a business-driven way. Together with the design methodology that is outlined in section 4, a business process and its functions can be used to define the services for the single layers. The boundaries of these services are determined by the business descriptions. This way, the principles of *autonomy* and *intersection points* (cf. [6]) are incorporated.
The use of aggregators promotes loose coupling. This is because the interaction necessary for the provisioning of a service is encapsulated by them. By including the layer of service coordination, this idea is incorporated into the reference architecture. It encapsulates application system-specific coordination from the process orchestration and includes loose coupling this way. The dedicated eventing system used to manage processes also decouples the components of a composite that is aligned with the reference architecture. By the means of the data exchange and data transformation layer, loose coupling is promoted by the means of validating the actual content of service interactions (cf. [5, p. 192]).
Finally, the data exchange and data transformation layer allows for another principle of service orientation: reusing the functionality of back-end systems.

3.1 Layers of the Reference Architecture

The introduced architectural framework provides several layers of abstraction. These abstraction levels loosely correspond to phases of a development methodology. The methodology that is described in section 4 is aligned with the presented framework.
According to the necessary steps for building composite applications, we identified five layers that composite applications should be build of. These layers are hierarchical in the way that lower layers provide functionality to the upper layers. An image showing all layers can be found in figure 1.

3.1.1 Layer Zero - Legacy Application Systems

The providers of functionality for composite applications are the various application systems that exist within the landscape of an organization. These application systems could just be a set of services that run on application servers. In such a scenario an organization is free to develop services as they are needed. However, this is rare in reality.
More likely application systems are COTS. Over time, organizations usually have bought and consolidated application systems for various purposes. Most likely these application systems are also integrated using Information-Oriented Application Integration. Of course, these landscapes support the processes of an organization well. They are not flexible, though. A BPIOAI approach would be preferable. However, it is simply because of budget that flexibility issues can not be addressed by reorganizing application systems into service providers.
This is why the application systems need to be capable of being integrated by exposing their functionality as services — services referred to in this thesis as "application services". Some proposals exist that aim to support the definition of application services in a way that required functionality and possibilities of application systems are considered as a trade-off (cf. [12], [9]). Despite the argument of budget, the idea of deploying COTS in an environment is to use standard software that is supported by the respective vendor. This is why it is usually not an option to deploy services on application systems at will (eg. by the means of wrappers as proposed by [12]).
If service orientation could be applied to a heterogeneous landscape, this is only possible if application systems remain unchanged (this means not reprogrammed but possibly reconfigured). The heterogeneity needs to be addressed elsewhere. This is why there are no constraints that can be put onto application systems. Furthermore, a flexible and independent mechanism is required that enables heterogeneous application systems to participate in a composite application. This mechanism is provided by the presented reference architecture.
Application systems are also considered as point of human-interactions with composite applications. This is because it is

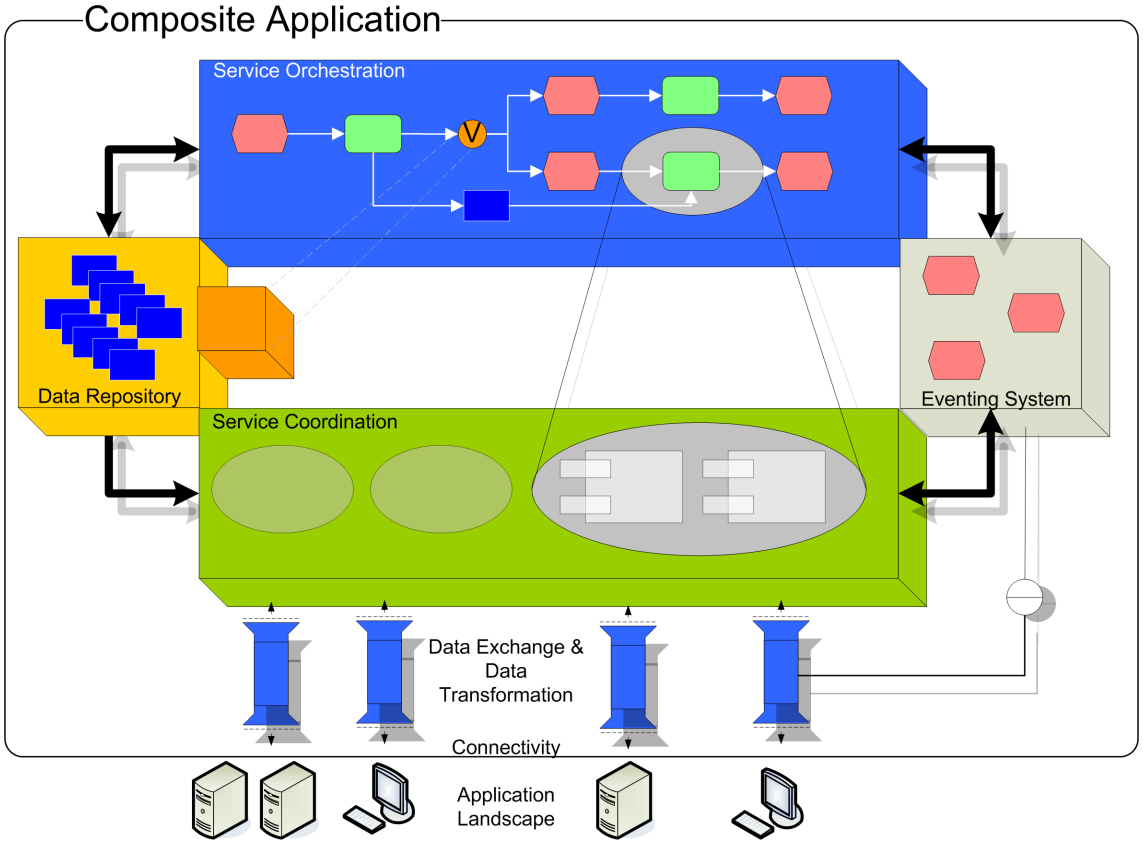

Figure 1. Reference Architecture

not feasible to remove all user interactions from application systems in order to establish a composite application.
This is a specific approach of this reference architecture. It has two additional advantages: first, it completely de-couples the control logic from user interactions or "wizards" that guide a user through a graphical user interface. Second, it allows for the transparent replacement of user interactions. This way the degree of automation can be increased without changing the overall process. Also outsourcing of certain tasks becomes easier.

3.1.2 Layer One - Connectivity

As a prerequisite, composite applications rely on a common protocol that is shared by service consumers and providers (cf. [3, p. 218]). The connectivity layer addresses the integration of application systems that do not provide functionality by using the common protocol of a specific composite application. It connects application systems and the composite application by homogenizing the protocol that is used to access functionality. The connectivity is realized by adapters. In general, an adapter "convert[s] the interface of a class into another interface [...]. An adapter lets classes work together that couldn't otherwise because of incompatible interfaces" [13, p. 139]. This description signifies that an adapter basically handles differences between two components by creating an intermediary abstraction between them. In the context of composite applications the definition of an adapter must be narrowed. Linthicum defines adapters from an application integration point of view as constructs that "[...] remove us from the need to deal with the interface details that communicate with a variety of different source and target systems. What's more, adapters provide more consistency from interface to interface because they are, by design, reusable from problem domain to problem domain. [...] They merely deal with the connectivity to the source or target systems" [3, pp. 23f.]. Hence, adapters are domain-independent intermediary components that connect in a system-specific way to application systems and to make the application systems accessible by composite applications. Adapters represent "[...] layers between the [composite application] and the source or target application" [3, p. 218].
Adapters address heterogeneous protocols and provide a translation that is specific for a certain (class) of application system. In addition to heterogeneous protocols, data formats are likely to differ in application systems. According to [14], data representation, data structures and data types can differ. Heterogeneous data is addressed by the data exchange and data transformation (DET) layer. In some scenarios, though, the platform of the DET might rely on a common data representation. In order to realize such scenarios, the adapters for the single application systems need to address differences in the data representation and transform the data prior to forwarding them to the DET. Even if it is conceptually possible, it is unlikely to deploy adapters and not a DET to connect to application systems. This is because application systems that do not natively support a service interoperability protocol usually do not store data using the data structures and data types used by a composite application.

3.1.3 Layer Two - Data Exchange and Data Transformation

One objective of this reference architecture is to standardize the way composite applications are built such that both the implementation of composite applications is eased and the conventional application integration is facilitated. This is achieved by the concept of integration flows. These integration flows describe a behavioral element on top of data-centric integration concepts and facilitate the realization of composite applications in the con-

text of heterogeneous application landscapes.

This second layer of the reference architecture is an optional layer that must be used whenever application systems do not stick to a globally defined data model, the appropriate data semantics or the required communication semantics. It unifies the data format of the connected application systems to a canonical data format (cf. [14, pp. 355-360] or [15]). It provides additional functionality for validity checking of data in terms of syntax and semantics as well as error handling procedures that need to be invoked whenever errors occur on this layer.[1] This way, integration aspects and business logic can be separated.

The data exchange and data transformation layer (DET) provides the functionality of an enterprise service (ESB) bus to a composite application. It mediates the business logic and the back-end application systems that are used to realize the business logic (cf. [16, p. 68]).
In [14], several so-called integration patterns are described and set into a relation. This relation is a basic pipe-and-filter architecture that describes a sequence of single patterns.
According to [8], basic services can be orchestrated by so-called micro-workflows in order to be orchestrated themselves by macro-workflows. This idea is combined with the integration patterns for the definition of the DET. In order to facilitate cross-system process orchestration, the DET describes a sub-set of the integration patterns that are orchestrated to so-called integration services. These services are, in turn, orchestrated by so-called integration flows.

In order to standardize and simplify the DET, all its functionality is encapsulated in so-called integration services (IS). These services are orchestrated by two different integration processes. One provides data to the upper layers of a composite — the integration in-flow (IIF). The other one publishes data from upper layers to the connected legacy systems. This process is called integration out-flow (IOF). Both the IIF and the IOF act as service providers: they expose the functionality of application systems to upper layers of the composite application as services.
As mentioned are the integration flows a replacement of the pipe-and-filter architecture that is used by [14] to categorize integration patterns for message based application integration. Additionally, they limit the applicable set of patterns to an extent that is required to realize composite applications in a heterogeneous application landscape. This way, business processes can be used to centralize the control the integration of application systems into composite application and prohibit the use of business logic inside the integration layer. The integration flows only handle the details of distributed and heterogeneous interactions.

3.1.3.1 Data Repository

In order to realize composite applications that utilize heterogeneous application systems, several aspects must be incorporated. This reference architecture introduces several layers that address these aspects. Consequently, if these layers span multiple platforms, a process executed by a composite application is distributed over all these platforms.
As the design methodology that is proposed in section 4 should be applicable to any actual target platform, this reference architecture, needs to be realizable with any arbitrary environment that follows the IT-strategy of the respective organization.
Multiple platforms are potentially required for realizing composite applications. Such platforms include application servers, integration servers and service orchestration tools. In order to apply the reference architecture to arbitrary combinations of such platforms, it needs to be ensured that all required platforms are able to collaboratively support business processes. As most of the architecture's elements operate on the processes' data, it is necessary that all elements have access to that data while ensuring acceptable performance and data consistency (cf. [5, pp. 185f.]). In order to allow this, the reference architecture introduces a global process context that is called the data repository. The data repository is the non-persistent[2] memory of composite applications. It keeps data referenced by events and makes it accessible to arbitrary components of a composite. This follows the idea of Kossmann that proposes to establish "a globally interconnected set of objects known as the ObjectUniverse, positioned in a huge address space referred to as the ObjectCosmos" [17, p. 1] in order to allow inter-operability in a distributed system. In the context of the different workflows used to realize a composite application, the data repository can be seen as a *blackboard* (cf. [18]) that is used in a workflow system (cf. [19]). It partially replaces the messaging paradigm with a spaces approach (cf. [20]).

The structure of the data stored in the data repository of a composite application is called *canonical data model* (CDM) (cf. [14, pp. 355-360]). A CDM provides an additional level of indirection between the single heterogeneous applications' individual data formats. If a new application is connected to a composite application, only transformations between CDM and the application's data format have to be realized, regardless of the number of applications that already participate (cf. [14, p. 356]). The CDM is a composite's own data format. If required, the transformation between the application's data formats and the CDM is realized by the data exchange and data transformation layer.
It is reasonable to use an already modeled and established data schema as the CDM. A good candidate for a CDM is the data model that used in the organisation's main Enterprise Resource Planning (ERP) system. Alternatively, global standards like UN/CFACT (cf. [21]) could be applied. A CDM is the data model for one composite application that usually supports one business process. Hence, if required, realizing multiple CDMs is possible. The usage of a CDM in a composite application does not introduce a company-wide data model.

The data repository acts similar to a tuple space (cf. [22]). Elements of the composite application get/read and store data to and from the data repository. These operations need to potentially be protected with transactions. Additionally, the data repository uses events to allow for smooth long-running transactions by already blocking requests that could interfere with already running processes.
Events determine both the appropriate processes to trigger and the set of data that is possibly affected by a certain event. Based on this idea relations between event types were introduced. These relations are used to describe consistency models for the data repository.

3.1.4 Layer Three - Service Coordination

From a top-down perspective the integration flows provide, in addition to the connectivity layer, a standardized mechanism for interacting with application systems. This is irrespective of communication or computational semantics and provides homogeneous data access as well.
As technical heterogeneity is addressed by the presented mechanisms, the services that are exposed by the integration flows can easily be orchestrated by using an orchestration engine. Without further concepts, however, the functionality that is provided by these services would be determined by the functionality offered by the application systems. Such services are aligned with neither the business requirements and nor the business tasks of a company. Hence, the business processes of an organization could either not be used to generate service orchestrations or the

[1] The error handling at this layer basically covers support procedures that need to be initiated whenever errors occur (human errors are mostly the cause of these errors).

[2] In order to address failure tolerance, the memory is persistent. However, the data store that is used is different from the owning application system that is used for storing the data

business processes would be restricted by the actual application systems. In order to apply the paradigm of service orientation and compose new business-centric functionality out of existent application functionality or to enrich functions within a specific context, it might be appropriate to combine the application specific functionality with new functionality. Expressed differently, application services might be needed to be aggregated to more problem-oriented services (*enterprise services*). The service coordination layer addresses this issue also referred to as *service mediation* (cf. [23]). It can be used in order to invoke two to *n* basic services using the integration flows in order to form the enterprise services that are then orchestrated. This is similar to the description of *business-driven service* pattern in [7] where services are orchestrated in a so-called micro-flows in order to correspond with the services that are orchestrated by a macro-flow. This composition of low-level services to *business-driven services* is considered rather static (cf. [7, pp. 44f.]).
[24] describes patterns that demonstrate how the gap between application services and a business process-centric orchestration can be dealt with using a so-called *process support layer* as a mediator. In particular, granularity problems and interdependency problems that prohibit the direct use of application systems from orchestrations are addressed. The *Composition*, *Decomposition* and *Bulk Service* patterns describe how differing granularity can be dealt with. The patterns *Sequentializing* and *Reordering* describe how interdependency problems can be addressed. Realizing such patterns is the purpose of the coordination layer.

In contrast to the *business-driven service* pattern of [7] and the *process support layer* of [24], the coordination layer is recursive. This means that an aggregated service that is composed at this layer might be aggregated again with services from this or lower layers to expose other high-level services. The benefit of this approach is that the aggregations themselves can remain flexible and their re-usability is increased.
A service coordination layer might aggregate both company internal and external services to services that are in turn usable both company internally and externally. This also introduces the need to support business protocols. These business protocols are sets of actions that have to be performed by multiple parties in order to allow successful execution of certain business functionality (cf. eg. [25]). They can be realized at the layer of service coordination.

An aggregation of services at the service coordination layer might also be required due to technical reasons. The service coordination layer exposes services to the service orchestration layer of a composite application. Such orchestrations are designed in alignment with business processes and not with "technical" constraints in mind. Whenever a multi-resource interaction is required that ensures consistent state transitions, this might be an indicator for the need of a technically motivated service aggregation. Consistent state transitions can be ensured by two means of transaction handling. Rather short-term transactions fulfilling the ACID properties by locking and rollback mechanisms or more long-term transactions without locking and with compensation actions. Of course, the orchestrated functionality is offered by the application systems and the consistent state transition is also assured by these systems.
Orchestrating the services of these systems does, however, raise the need for a cross-service transactional coordination. Even if the application systems have to support the transactional coordination by appropriate compensation operations and/or by supporting transactional protocols, the coordination itself has to be controlled at this layer of the composite application. The service coordination layer controls distributed transactions by initiating them and passing the transactional context to application systems (possibly via the DET) as well as to the data repository.
According to [26], transactional coordination can consist of two layers. One layer for so-called *local transactions* with ACID properties and one for *global transactions* with relaxed transactional properties. The latter one uses ACID transactions as black-boxed functionality to form long-term global transactions. By distinguishing these two layers of transaction handling, the idea of separating concerns of different transactional properties is incorporated.
Composite applications that are implemented using the presented reference architecture realize local transactions at the service coordination layer as it acts as the controlling instance for ACID transactions .
Meeting the long-term characteristics of global transaction, the isolation and atomicity properties can be relaxed and so-called safepoints can be used (cf. [26]). Relaxing the isolation property is realized by publishing intermediated results to the global context (for the presented architecture this is the data repository). Atomicity is relaxed by introducing compensating transactions that "undo" other transactions. Both context publication and compensation transactions are local transactions.

Safepoints are local transactions that are marked by this special property of being a safepoint. Thus, the fundamental support for global transactions is formed by local transactions. This point of view is in line with the concept of a recursive service coordination layer.
[26] also proposes a way to specify transactional properties (such as the safe point properties for local transactions) and an execution model that supports global transactions based on these specifications. This execution model dynamically calculates workflow paths for partial or complete compensation of global transactions, if required. At the given point this is, however, not seen as a mandatory feature for a composite application. This is why only the notion of short-term and long-term transactions and the notion of compensating transactions are considered a necessity for composite applications.
Another necessity for deploying a service coordination layer is to realize complex interaction patterns. A DET offers means supporting most of the known service interaction patterns. However, the *contingent request* can not be implemented by solely using the DET. The service coordination layer is required here as well.

3.1.5 Layer Four - Business Processes

The BPIOAI approach of [3] describes that the control flow that executes distributed functionality should be designed with reference to a business process. Since one major benefit of the service-oriented architectural style is the centralization of the control over distributed functionality, the aim of this reference architecture is to allow for a control flow that is described by the means of business processes. This way control is not only centralized but also aligned with business requirements. Only small technical constraints should prohibit the direct deployment of business process descriptions. The place within the reference architecture to deploy these processes is the *Business Process Orchestration Layer*.
Business processes can be described by workflows in an imperative way using a workflow description language. Workflows have several aspects or perspectives that together form the description of a workflow. These perspectives are the control flow, data, resource and operational perspective (cf. [27]). The control flow "describes activities and their execution ordering through different constructors, which permit flow of execution control, e.g. sequence, choice, parallelism and join synchronization" [27, p. 2]. The data perspective describes business and processing data of the workflow as well as pre- and post-conditions for the tasks of the workflow. The resources and the operational perspective describe how workflows are executed in terms of their organizational support and of supporting application systems.
The *Business Process Orchestration Layer* of a composite application should consist of two elements. First, a workflow engine that executes the control flow by orchestrating application services and service coordinations is required. There a business

process is deployed as the central control flow in a composite application.
Second, decisions within the control flow should be controlled by a dedicated `Decision Service` that operates on the data of such a workflow. It is necessary to define a dedicated service as the process context is kept outside the actual workflow engine. These two elements of the process orchestration layer are described in the following.

3.1.5.1 Workflow System for Service Orchestration

The workflow system for service orchestration is the part of the *Business Process Orchestration Layer* that provides a runtime-environment to execute workflows that coordinate services following the control flow of an actual business process. The orchestration needs to be described in a workflow description language that is deployed to the workflow system. However, the actual design and validity of such processes must be checked during design-time.
The services being executed are provided by application systems that are possibly mediated by integration flows of the DET and the service coordination layer. The (data) context of the process is kept in the data repository (cf. section 3.1.3.1). This is because of architectural considerations that involve consistency and simplification but also the simple necessity to expand a process' context throughout a composite application. The workflow system that is used for service orchestration does therefore solely invoke external service providers. In order to realize conditional expressions, the workflow engine utilizes `Decision Services` that are connected with the data repository. Such services are used to decide on conditional expressions in a given context.
Business processes do not only involve application systems, though. Human interaction is often also part of such processes. From a software architecture point of view, human interactions are realized as the interaction with back-end systems. Hence, humans are considered service providers that use a user-interface to receive input and provide output of a certain functionality. As a consequence, there are no constraints imposed to the layer of process orchestration.

3.1.5.2 Decision Service

The layer of process orchestration forms the central component that controls the overall execution of a composite application. The actual decisions within the control flow (exclusive choice, multi-choice) and structural workflow patterns that require such decisions (e.g. exit conditions in loops) are often based on the context of a certain process instance.
Workflow descriptions that are used in such service orchestrations are monolithic blocks that usually involve several business rules. A business rule "is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control the behavior of the business" [28, p. 30].
Managing or changing the single rules that are embedded in such blocks is both difficult and time-consuming (cf. [29]). In order to increase the maintainability of composite applications that apply the presented architecture beyond the possibilities of the service-oriented architectural style, business rules are managed by `Decision Services`. They are used to determine the actual control flow of a service orchestration. According to the classification of [29], a `Decision Service` decides on *reaction rules* for a business process. It uses the data repository to check whether certain conditions apply. Based on the output of the `Decision Service`, the service orchestration may invoke different services.
By using a `Decision Service`, the business logic that underlies such decisions can be described independently from the used business process description language in a separate service. This way, the decision logic becomes reusable.
In order to reduce complexity, handle transactions and allow for a multi-layered architecture, a generic data description kept inside the process environments by the means of the `Data Repository` is part of this reference architecture. Since the constructs proposed are quite complex and independent from a certain process orchestration language, it could occur that an actual language is not capable of using the generic business data. A `Decision Service` also provides an interface from the service orchestration layer to the `Data Repository` to increase the possible platforms with which the reference architecture could be realized.

According to [30], a business rules engine consists of a *rule base*, a *working memory*, a *pattern matcher* and an *inference engine*. "The *working memory* holds the data on which the rule engine operates" [30, p. 36]. In the reference architecture, this data is kept in the data repository. The *rule base*, the *pattern matcher* and the *inference engine* internal components and their specification is out of the scope of the reference architecture for composite applications.
From an architecture point of view, a `Decision Service` needs to be able to decide *reaction rules*. The actual reaction to such a rule is, however, performed by the workflow engine as the central control instance of a composite application.[3]

The patterns we assign for this layer are the Enterprise Integration Patterns by Hohpe and Woolf [14]. In order to structure these patterns we introduced integration flows that form a taxonomy for the integration patterns [31]. This taxonomy describes standard services that are required to read or write data to application services, respectively. In addition to the functionality provided by the integration patterns, the integration flows additionally deal with data heterogeneity and communication semantics. This why optional constructs are incorporated into the generic integration flows. Such constructs are for instance optional acknowledgments that are triggered in certain states of the integration flow in order to support certain communication semantics. This optional functionality is considered as design pattern as well. All relevant patterns and the categorizing taxonomy are described in [31].

4. Composite Application Design – A Step-by-Step Process

The section outlines an integration methodology that leverages the described reference architecture to build composite applications based on a business process. The approach consists of 16 steps that outlined in figure 2. If performed correctly, these steps describe how the system design for an actual use case can be derived.
The methodology combines a top-down approach with a bottom-up approach. The top-down approach takes the requirement descriptions as its input. Emphasis is placed on deriving services.
After refining the derived service candidates, the link between the design and the actual constraints of an application landscape is performed. From this point forward, a bottom-up approach incorporates the identified constraints into the design of the single components of the reference architecture for composite applications. By doing so, the initial description of the business process is changed in a way that it can be used as the central service orchestration.
The single steps are described as a sequence. However, iterations of certain steps or the reworking of complete branches is possible.

Step 1: List all Business Process Activities

The input of the methodology is a complete description of the business process that needs to be realized using a composite application. Based on the described process, the single services that are required by the reference architecture can be derived. This procedure of deriving the services is based on the understanding

[3] If the rules engine is integrated with the workflow engine as it is described by [29], this distinction is blurred.

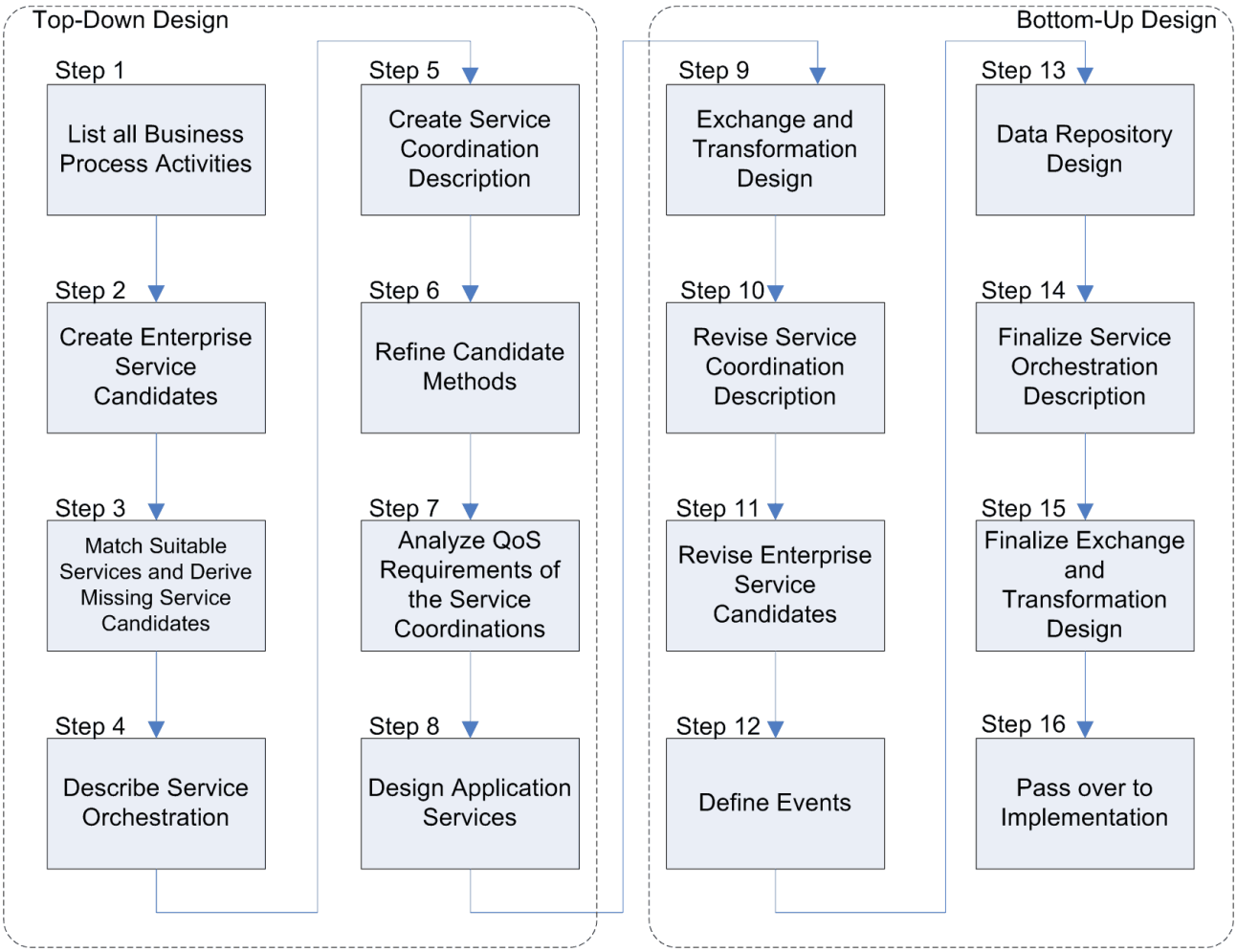

Figure 2. Steps of the Design Methodology

of a business process as a possible service orchestration and the business tasks as orchestrated enterprise services.
Thanks to the reference architecture, the enterprise services can be considered providers of the functionality required by business tasks. The single business tasks are described in the business process description. In contrast to Erl's methodology that begins by refining the actual business process (cf. [9, pp. 397-430]), this approach takes the activities of a business process as an input for later phases. Each function of a business process' control flow is considered to be supported by an enterprise service. This is why it is necessary to extract the description of the respective process' functions to identify these enterprise services.

The output of this step of the methodology is a list of business functions that includes the name of the function, the input and output data of the function and a rough, informal description of the task. If applicable, the back-end application system that is currently used for such tasks (eg. order processing) can also be indicated.

Deliverables: Process functions; data flow for each function; informal description of each identified function.

Step 2: Create Enterprise Service Candidates

In this second step, the list of business functions is used to describe enterprise service candidates. If a process was not designed using existent enterprise services and if no services that fit a process description can be found within the actual service registry (i.e., no "perfect matches"), they need to be created.
Usually new enterprise services are built out of existing functionality as well as out of new service methods that are designed using the following steps (especially step three). The composition that realizes an enterprise service by using new and existent (coordination) services is called service coordination. The coordination services it consists of can only be composed in a way that supports the enterprise services while not being enterprise services themselves. The actual design of the enterprise services is therefore a prerequisite for the design of coordination service methods and the respective service coordination.
A business process can be seen as an event-driven computation of data (cf. [32]). The events do not only determine the task that should follow an event, but also the data that is transmitted during this event. This data is both context data of the process as well as output from preceding enterprise services. Both the events and the data are described in the model of the business process. The data is described in greater detail in the according data perspective of a business process.
As the reference architecture and this methodology were created to directly allow the use of business processes as service orchestration, the single tasks of the business process are considered to be enterprise services. Deriving these enterprise services is simple. Each business function identified in step 1 is considered an enterprise service. The input for this method is the data that is consumed by the business task. The result of a business task is

the output of the enterprise service method.[4]

Deliverables: List of coordination service candidate methods; list of coordination service methods that need to be composed; functional requirements for all service candidates.

Step 3: Match Suitable Service Methods and Derive Missing Service Method Candidates

The service design methodology of [9] asserts that services which are used in a process orchestration should fit into the orchestration. In the given context, "fitting" refers to the fact that the service computes a set of input parameters and then provides the described functionality and the required output parameters (cf. [9, pp. 205ff.]). Services that fit into an orchestration can be identified by using the deliverables of step 1 and 2 (process functions; data flow for each function; informal description of each identified function). If, however, no matching services can be identified, it is an informal task to derive the missing services for the identified process functions (cf. [9]). However, this is a complex task and it's description has to be omitted here.

Deliverables: List of coordination service candidate methods; list of coordination service methods that need to be composed; functional requirements for all service candidates.

Step 4: Describe Service Orchestration

As service orientation is a paradigm for control centralization over distributed functionality, the control flow logic needs to be described. According to [9], potential types of logic that need to be considered are:

- "business rules
- conditional logic
- exception logic
- sequence logic" [9, p. 403]

Part of this service orchestration logic is typically included in the initial business process model. This is especially true for the sequence and conditional logic. These parts describe how the preconditions of service invocations can be met, how data is exchanged between services and how post-conditions of services should be computed.
"Business rules are statements about how business is conducted, i.e. the guidelines and restrictions with respect to business processes in an enterprise" [33, p. 9]. Exception logic finally describes how a process should react on exceptional states. Exception logic is normally both part of a business process and part of a more general exception handling procedure. A process description will usually contain a procedure that will be used to react to business exceptions that are closely related to the process. However, the general part of exception rules describes how exceptions, that are not handled directly by the process, should be treated. Reasons for not including such rules into processes are that they were not foreseen in the description (e.g. technical routing errors while calling a service).

Deliverables: Sound description of the process orchestration; conditional, exception logic and sequence logic; business rules

Step 5: Create Service Coordination Description

From the initial requirements, a service orchestration (with associated rules) as well as method candidates that act as the single service providers for the process orchestration were derived in the previous step. In order to further drill-down the abstraction level and realize the business process implementation, every single enterprise service needs to be described in detail.
This description includes the already known interface of the actual enterprise service as well as the coordination service method candidates that were identified in step 3. In this step of the methodology, the identified candidate methods are composed in a way that the composition provides the realization of the respective enterprise service. Usually, the coordination should be a sequence of all coordination service methods in a way that required data is produced before it is consumed. However, either due to the modeling of the initial business process or due to special requirements, the description of a service coordination might include conditions or parallel tasks, too. This is why the creation of the coordination description is considered to be a manual (yet simple) step. In conjunction with business process modeling rules, this step could be automated. This might be realized by a simple, data dependency-based approach (that simply links services in a sequential way). An alternative would be to apply more sophisticated automated service composition approaches. [33] is an example for such an automated composition process. More general information about the requirements for automated service composition is given by Meyer et al. in [34].

Deliverables: Specification of the service coordination for every enterprise service using the candidate methods that were identified in step 3.

Step 6: Refine Candidate Methods

Depending on the outcome of the previous steps, the requirement to refine the candidate methods of step 3 might arise. Basically, this is a step the designer of a composite can perform in order to manually adjust the shape of the service method candidates.
One possible scenario for the need to adjust the candidate methods is that the assumed interface does not fit well together with the required service coordination. In particular, entity-specific creation methods might not be required for creating new entities. Furthermore, these methods could include manual lookups with human interaction. Whenever a human agent provides a required functionality of a method, context information is needed. Hence, an example for an adjustment in this step is adding context information to entity-specific candidate methods.
Another possible requirement for the redesign of candidate methods is an inefficient design of a service coordination. Such design efficiency can be measured by the design metrics described in [35].

Deliverables: A list of stateless, named and probably revised coordination service method candidates.

Step 7: Analyze QoS Requirements of Service Coordinations

The previous steps of this methodology produced a service orchestration that orchestrates enterprise service (candidates). Additionally, so-called service coordinations, that describe how coordination services can be used to realize the single enterprise services, were initially defined. In this step of the methodology the quality-of-service (QoS) requirements of the single service coordinations are analyzed.
An element that must be considered is the manner in which each service coordination for the single enterprise services deals with the unavailability of the composed services. Basically, this involves the notion of distributed ACID transactions as well as the definition of transactions with relaxed ACID properties.
ACID transactions are defined as a set of service methods of a service coordination that need to be executed together or not at all.
Transactions can also be relaxed in terms of their atomicity and isolation properties. In order to utilize relaxed, global transactions, (sets of) method calls can be marked as being *safepoints*. The return values of such method calls must be persisted in the

[4] As methods are considered to be ordered sets, a method can have multiple return parameters. How this is actually realized is not the concern at this point.

data repository before subsequent services are called. In case of exceptions during subsequent steps, these steps need to be compensated. Processing is then restarted after the most recent safepoint with the data that was previously committed by that safepoint method to the data repository.
If safepoints are identified, compensating actions for a set of methods also need to be defined. In order to define such actions, a group of services and different exceptions, that trigger them, are defined. Possible types of exceptions include work item failures, deadline expirations and resource unavailabilities (cf. [36]). In order to realize a compensating activity, a corresponding service method has to be identified for each defined exception. Compensating operations that work on the data repository should be realized as services at the service coordination layer. Additional functionality that might be required needs to be realized as ordinary application services that are part of such an aggregation.
Service coordinations as a whole are always considered safepoints. Hence, they also have to commit their result to the global data repository. The compensating activities for the enterprise services should be defined as part of the business process model.

Deliverables: Sets of coordination service methods for distributed transactions; sets of methods that are marked as safepoints; sets of methods that are to be compensated; method descriptions for realizing compensations; required maximal failure rates for the coordination services for all enterprise services.

Step 8: Design of Application Services

The functionality of the coordination services often already exists within an application system(s). If so, four options for utilizing them in a composite application exist. Those are the direct (re-)use of application systems' functionality, mediated reuse of such functionality, direct use of wrappers that are implemented within the application systems and mediated access. The latter is necessary whenever the implementation of wrappers is prohibited.
If only part of the required functionality can be identified in application systems, additional services will be required. These additional services are added to the list of required coordination services together with coordination services for the functionality that was identified.
According to the identified functionality, previously designed coordination services might be changed or adjusted, too. The quality-of-service requirements should also be identified for the revised service methods.

Deliverables: For *direct reuse*, the respective service methods need to be determined. For the *direct use* of services, the not yet created application services must be described.
For mediated reuse, suitable functional modules need to be identified. Their use must also be described. Further, suitable connectivity options are required and potential new services must be described as services for *direct use*.
If *direct use of wrappers* is applicable, the wrappers need to be specified.
If application systems can only be used via *mediated access*, the data scheme that should be accessed needs to be described. Additionally, the available connectivity options need to be determined.

Step 9: Exchange and Transformation Design

Integration flows are required for any method of an application system that needs to be mediated. As the realization of the actual application services and their combination with the DET are usually heavily constrained by technical circumstances, steps 9 and onward need to be performed while taking the actual target platform and application systems into consideration. This is how the platform-independent design that is created in the first steps is aligned to an actual platform.

In order to design the necessary integration flows, the required interactions with the service providers first need to be identified. This is achieved by specifying the appropriate service interaction pattern (cf. [37]) and its respective design decisions. Based on the identified pattern, the required integration flows, as well as some integration services, can be identified. The identified mapping indicates the required integration services, as well as several design decisions for each integration service.
Based on the identified integration services, the design needs to be completed for each service. These service form a pattern taxonomy that is described in-depth in [31].

Deliverables: Complete specification of the integration flows for every mediated service interaction. The description needs to include the specifications for the necessary integration services.

Step 10: Revise Service Coordination Description

The design of the actual application systems as well as the design of the mediating integration flows reveals the applicability of the top-down design for the coordination services. Based on the deliverable of the two previous steps, the coordination descriptions might be changed. If changes occur, the quality-of-service requirements need to also be analyzed for the new coordination.

Deliverables: Final description of the single coordination services as well as the service coordination that aggregates the coordination services to enterprise services.

Step 11: Revise Enterprise Service Candidates

During the step of redesigning service coordinations, technical or organizational constraints that prohibit the realization of enterprise services that are aligned with the actual business process might be identified. If this is the case, two possibilities for proceeding exist.
First, the whole process can be started from the beginning. This is a preferable option if the modeled business process does not fit into the landscape of an organization. The identified constraints should then be used in order to define a business scenario that is in alignment with these constraints.
The second option is to revise the actual design of enterprise services in a way that the defined service coordinations can be realized. This will usually involve defining additional input and/or output parameters for the single enterprise services.
If the enterprise services are changed this is likely to also impact the process orchestration. If the enterprise services are changed, the service orchestration will need to be adjusted in a later step (step 14). It must be noted that changing a process orchestration is a violation of the objective of aligning application development with business needs.

Deliverables: Decision about whether to continue the design or to start over. If the design is continued, the revised enterprise services must be defined. If the decision is to start again, the constraints that led to this decision are required deliverables.

Step 12: Define Events

At this phase of the design, the enterprise services are defined. In addition, how these services can be realized and what transactional requirements exist is described.

Deliverables: Event types and their respective boundaries; relations between event types; and optionally, the definition of coordination services that pass data between different scopes.

Step 13: Data Repository Design

Based on the `Event Types` that were identified in the previous step, the preconditions for every event type can be defined. Such preconditions might exist in terms of data existence. They describe that an event of a certain type can only be computed if certain data is stored in the data repository of a process. Such

preconditions need to be identified in order to appropriately configure a data repository.
On top of data prerequisites, data types from the data perspective need to be incorporated into the design of the data repository (that addresses the principle of "interface reference" of [6]). This is achieved by defining the smart proxy for the given process as well as the transfer objects that are used to access the smart proxy. This information can be found by analyzing the data model of the business process. The business rules that are identified in step 4 should also be analyzed as the data by which they are defined is required to be kept in a data repository. Another input to the definition of smart proxies and transfer objects are the interfaces of the coordination services. The data that is required by those services also needs to be integrated into the data repository and made accessible by the smart proxies.
Based on the identified data and the respective transfer objects, the data objects can then be grouped by the event types that concern them. Together with the event type relations from the previous step, the data repository configuration can thus be completed.

Deliverables: Design of smart proxies and data transfer objects; configuration instructions for the data repository.

Step 14: Finalize Service Orchestration

The deliverables of the previous steps describe all the facets of a composite application. This description is aligned with the description of the business process. However, several constraints could require changing the service orchestration.
If a service coordination cannot be realized as required, this also impacts the service orchestration. Such changes need to be performed during this step. As such requirements for changing the orchestration are usually imposed by application system-specific constraints, the adjustment of the orchestration is not structured further.

A mandatory activity of this step is the design of the necessary `Decision Services`. Based on the business rules that were identified in step 4 and the available data that is represented by the design of the data repository, the rules need to be described and stored into a `Decision Service`.

Deliverables: Business rules that are formulated such that they can be interpreted by the actual `Decision Service(s)`; decision regarding the method of integrating `Decision Service(s)` in a platform-specific way; revised service orchestration that incorporates communication with the `Decision Service(s)`, `Event Service` and service coordinations for data passing; all necessary changes to the orchestration that are necessary due to informal constraints also need to be reflected.

Step 15: Finalize Exchange and Transformation Design

This step of the procedure is required in order to describe the `Trigger Services` that are needed to mediate service interactions. `Trigger Services` are used to extrinsically invoke composite applications.
Based on the event types and the data repository design, the data that is required as a prerequisite for a process is known. By analyzing the actual interaction, the source of this data can also be identified. If the source of such data is an agnostic application system, a `Trigger Service` needs to be integrated. By adapting to the interface of the service consumer (the application system) the interface of the respective `Trigger Service` is determined. Based on the identified event type, the `Heterogeneity Service` of the respective `Trigger Service` can also be designed. Additionally, filter logic needs to be described. This logic is determined by the scenario as well as by technical constraints of the service consumer invoking a `Trigger Service`. Finally, based on the data repository design, the *write* activity of a `Trigger Service` that stores the data into the data repository can be defined.

Deliverables: Finalized description of the integration flows including the design of all necessary `Trigger Services`.

Step 16: Pass over to Implementation

After step 15, the design of the composite applications and the services they consist of is finished. The design is described in a platform independent manner. Additionally, some constraints of the target platform are also incorporated.
Based on this design, a composite application can subsequently be implemented. It is not recommended to add additional design artifacts for this phase. In contrast to the idea of a Model Driven Architecture (cf. [38]), this methodology does not aim to transform platform independent design into executable, platform-specific applications.

5. Summary and Outlook

The presented reference architecture is aligned around the notion of *abstraction*. It puts a business process at the center and uses *loosely coupled* and *autonomous* services to establish a link between the actual business process and the heterogeneous application landscape. The link to heterogeneous application systems is facilitated by the means of a flexible integration layer. By emphasizing the reuse of application functionality, service orientation is introduced as a beneficiary style for application integration that allows for realizing adaptive applications in established and stable system landscapes. This way, the reference architecture becomes more applicable for organizations that aim at using standard software.

In order to make the reference architecture applicable in actual projects, a design methodology for composite applications was presented. This methodology combines a top-down and bottom-up approach so as to translate an actual business process into the design of a composite application that is based on the described reference architecture.
Together, these concepts facilitate the application of SO for large organizations. As such, these concepts can be seen as an enabler for advanced service-oriented principles in this context. Especially semantic service provisioning (cf. [11]) is a concept that can be seen as a next step after the application of basic service-oriented principles.

6. References

[1] Anil Nori and Rajiv Jain. Composite applications: Process based application development. In Alejandro P. Buchmann, Fabio Casati, Ludger Fiege, Meichun Hsu, and Ming-Chien Shan, editors, *TES*, volume 2444 of *Lecture Notes in Computer Science*, pages 48–53. Springer, 2002.

[2] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE*, pages 179–185, 1995.

[3] David S. Linthicum. *Next Generation Application Integration*. Addison-Wesley, Boston, MA USA, 2004.

[4] Mike P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *WISE*, pages 3–12. IEEE Computer Society, 2003.

[5] Gregor Engels, Andreas Hess, Bernhard Humm, Oliver Juwig, Marc Lohmann, Jan-Peter Richter, Markus Vo, and Johannes Willkomm. *Quasar Enterprise - Anwendungslandschaften serviceorientiert gestalten*, volume 1. dpunkt Verlag, 2008.

[6] Joachim Schelp and Robert Winter. Towards a methodology for service construction. In *HICSS*, pages 64–70. IEEE Computer Society, 2007.

[7] Carsten Hentrich and Uwe Zdun. Patterns for process-oriented integration in service-oriented architectures. In *Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006)*, 2006.

[8] Uwe Zdun, Carsten Hentrich, and Schahram Dustdar. Modeling process-driven and service-oriented architectures using patterns and pattern primitives. *ACM Trans. Web*, 1(3):14, 2007.

[9] Thomas Erl. *Servcie-Oriented Architecture*, volume Fourth Printing of *The Prentice Hall service-oriented computing series*. Prentice Hall, Inc., Upper Saddle River, NJ USA, February 2006.

[10] Ali Arsanjani and Abdul Allam. Service-oriented modeling and architecture for realization of an SOA. In *IEEE SCC*, page 521. IEEE Computer Society, 2006.

[11] Domini Kuropka, Peter Trger, Steffen Staab, and Mathias Weske, editors. *Semantic Service Provisioning*. Springer, Berlin, Germany, 2008.

[12] Willem-Jan van den Heuvel, Jos van Hillegersberg, and Mike P. Papazoglou. A methodology to support web-services development using legacy systems. In *Proceedings of the IFIP TC8 / WG8.1 Working Conference on Engineering Information Systems in the Internet Context*, pages 81–103, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1996.

[14] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns*. The Addison Wesley Signature Series. Pearson Education Inc., 2004.

[15] G. Kaufman. Pragmatic ecad data integration. Technical Report 1, New York, NY, USA, 1990.

[16] Colombe Herault, Gael Thomas, and Philippe Lalanda. Mediation and enterprise service bus: A position paper. In *Proceedings of the First International Workshop on Mediation in Semantic Web Service (MEDIATE) 2005*, 2005.

[17] Rainer Kossmann. An architectural framework for semantic inter-operability in distributed object systems. In J. Sutherland, D. Patel, C. Casanave, G. Hollowll, and J. Miller, editors, *Business Object Design and Implementation*. Springer- Verlag London, 1995.

[18] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: A system of patterns*. John Wiley & Sons, Inc. New York, NY, USA, 1996.

[19] Stefan Kleine Stegemann, Burkhardt Funk, and Thomas Slotos. A blackboard architecture for workflows. In Johann Eder, Stein L. Tomassen, Andreas L. Opdahl, and Guttorm Sindre, editors, *CAiSE Forum*, volume 247 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[20] Daniel Martin, Daniel Wutke, Thorsten Scheibler, and Frank Leymann. An EAI pattern-based comparison of spaces and messaging. In *EDOC*. IEEE Computer Society, 2007.

[21] Birgit Hofreiter, Christian Huemer, and Klaus-Dieter Naujok. UN/CEFACT's business collaboration framework - motivation and basic concepts, 2004.

[22] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[23] Liangzhao Zeng, Boualem Benatallah, Guo Tong Xie, and Hui Lei. Semantic service mediation. In Asit Dan and Winfried Lamersdorf, editors, *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 490–495. Springer, 2006.

[24] Gero Decker. Bridging the gap between business processes and existing it functionality. In *Proceedings of the First International Workshop on Design of Service-Oriented Applications (WDSOA'05)*, 2005.

[25] *RosettaNet Partner Interface Processes*.

[26] Paul Grefen, Jochem Vonk, and Peter Apers. Global transaction support for workflow management systems: from formal specification to practical implementation. *The VLDB Journal*, 10(4):316–333, 2001.

[27] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[28] Anis Charfi and Mira Mezini. Hybrid web service composition: business processes meet business rules. In Marco Aiello, Mikio Aoyama, Francisco Curbera, and Mike P. Papazoglou, editors, *ICSOC*, pages 30–38. ACM, 2004.

[29] Florian Rosenberg and Schahram Dustdar. Business rules integration in BPEL - a service-oriented approach. In *CEC*, pages 476–479. IEEE Computer Society, 2005.

[30] Christoph Nagl, Florian Rosenberg, and Schahram Dustdar. VIDRE - a distributed service-oriented business rule engine based on RuleML. In *EDOC*, pages 35–44. IEEE Computer Society, 2006.

[31] Helge Hofmeister and Guido Wirtz. A Pattern Taxonomy for Business Process Integration Oriented Application Integration. In Kang Zhang, George Spanoudakis, and Giuseppe Visaggio, editors, *SEKE*, pages 114–119, July 5-7 2006.

[32] Sonia Lippe, Ulrike Greiner, and Alistair Barros. A survey on state of the art to facilitate modelling of cross-organisational business processes. In M. Nüttgens and J. Mendling, editors, *Proceedings of the 2nd GI Workshop XML4BPM*, pages 7–22, March 2005.

[33] Liangzhao Zeng, Boualem Benatallah, Hui Lei, Anne H. H. Ngu, David Flaxer, and Henry Chang. Flexible composition of enterprise web services. *Electronic Markets*, 13(2), 2003.

[34] Harald Meyer and Dominik Kuropka. Requirements for automated service composition. In Johann Eder and Schahram Dustdar, editors, *Business Process Management Workshops*, volume 4103 of *Lecture Notes in Computer Science*, pages 447–458. Springer, 2006.

[35] Helge Hofmeister and Guido Wirtz. Supporting service-oriented design with metrics. In *Proceedings of the 12th IEEE International EDOC Conference (EDOC 2008)*, October 2008.

[36] Nick Russell, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede. Workflow exception patterns. In Eric Dubois and Klaus Pohl, editors, *CAiSE*, volume 4001 of *Lecture Notes in Computer Science*, pages 288–302. Springer, 2006.

[37] Alistair P. Barros, Marlon Dumas, and Arthur H. M. ter Hofstede. Service interaction patterns. In Wil M. P. van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management*, volume 3649, pages 302–318, 2005.

[38] Richar Soley. Model Driven Architecture. Whitepaper, November 2005. Last access: 20th December 2007.