A Survey of Patterns for Service-Oriented Architectures

Uwe Zdun

New Media Lab, Department of Information Systems
Vienna University of Economics and Business Administration
Augasse 2-6
1090 Vienna, Austria
e-Mail: zdun@acm.org

Carsten Hentrich

SerCon GmbH - IBM Business Consulting Services
c/o IBM Deutschland GmbH
Hechtsheimer Str. 2
55131 Mainz, Germany
e-Mail: chentric@de.ibm.com

Wil M.P. van der Aalst

Department of Information Systems
Faculty of Technology and Management
Eindhoven University of Technology
PO Box 513, NL-5600 MB Eindhoven, The Netherlands
e-Mail: w.m.p.v.d.aalst@tm.tue.nl

Abstract: Service-Oriented Architectures (SOA) are a promising means to integrate heterogeneous systems, but virtually no technology-neutral approach to holistically understand SOAs exists. We tackle this problem by introducing a survey of technology-independent patterns that are relevant for SOAs, and are working towards a formalised pattern-based reference architecture model to describe SOA concepts.

Keywords: Patterns, Pattern Languages, SOA, Service-Oriented Architecture, Reference Architecture

References to this paper should be made as follows: Zdun, U. and Hentrich, C. and van der Aalst, W.M.P. (2005) 'A Survey of Patterns for Service-Oriented Architectures', *Internet Protocol Technology*, Vol x, No x, pp.x—x

Biographical notes: Uwe Zdun is working currently as an assistant professor in the Department of Information Systems at the Vienna University of Economics and Business Administration. He received his Doctoral degree from the University of Essen in 2002. His research interests include software patterns, software architecture, scripting, object-orientation, AOP, and Web engineering. Uwe has published in numerous conferences and journals, and is co-author of Wiley's Remoting Patterns book. He has participated in a number of R. & D. projects and industrial projects. Uwe is (co-)author of the object-

oriented scripting language Extended Object Tcl (XOTcl), the Web object system ActiWeb, and many other software systems. He acts as a reviewer in journals and conferences. He has co-organized a number of workshops at conferences such as EuroPLoP, CHI, and OOPSLA. Uwe serves as conference chair for EuroPLoP 2005.

Carsten Hentrich is a senior consultant at the IBM IT consulting subsidiary SerCon in Germany. He has seven years of professional management consulting experience as a business consultant and software architect, especially in the fields of business process management and workflow management. He holds an MSc with Distinction in Software Engineering from Oxford University and a degree in Computer Science from the University of Applied Sciences in Wiesbaden, Germany. He is currently undertaking extra occupational doctoral research on a part-time basis at Eindhoven University of Technology.

Wil van der Aalst is a (full) professor at the Information Systems (IS) department of the Faculty of Technology Management (TM) of Eindhoven University of Technology (EUT). He is also an (adjoint) professor at the Centre for Information Technology Innovation (CITI) of Queensland University of Technology (QUT). His research and teaching interests include information systems, workflow management, Petri nets, specification languages, and simulation. Since March 2000, he is head of the IS department. Before he was head of the Specification and Modeling of Information systems (SMIS) research group of the Computing Science department of EUT.

1. Introduction

This paper aims at providing an architectural framework for Service-Oriented Architectures (SOA) in form of a survey of patterns relevant for building SOAs. Though built on similar principles, SOA is not the same as Web services, which indicates a collection of technologies, such as SOAP and XML. SOA is more than a set of technologies and runs independent of any specific technologies. A service-oriented architecture is essentially a collection of services that are able to communicate with each other [4]. Each service is the endpoint of a connection, which can be used to access the service and interconnect different services. Communication among services can involve only simple invocations and data passing, or complex coordinated activities of two or more services. In this sense, service-oriented architectures are nothing new.

However, SOAs are not well-defined at the moment and there is not much architectural guidance how to design a SOA – many definitions and guides are focused on concrete technologies, not on the essential elements of the architecture. To overcome this problem, we propose a reference architecture based on software patterns. Software patterns provide reusable solutions to recurring design problems in a specific context [2, 8]. In this paper, we use software patterns because they abstract from concrete, technology-dependent solutions and they provide timeless, proven solutions. The goal of our pattern survey is to help in understanding the principles, key constituents, and key structures of a SOA, apart from any concrete technology.

We use the pattern-based approach to enable a broad, platform-independent view on SOAs that still contains all relevant details about the technical realization alternatives. The main contribution of this paper is to provide a holistic, architectural approach to guide the design of SOAs.

To reach these goals we adapt software patterns from different sources that were described originally in a number of different domains, such as remoting [32], messaging [15], resource management [17], networked and concurrent objects [27], software architecture [5], component integration [35], object-oriented design [13], e-business [1], process-driven architectures [14], business objects [12], and workflow systems [31]. A major contribution of our work is the domain-specific combination of these patterns – in the SOA domain. We explain these patterns very briefly, where they appear first. For full pattern descriptions please refer to the original pattern descriptions that are referenced.

2. Basic Service Architecture

The basic concept of a service-oriented architecture (SOA) is quite trivial: a service is offered using a remote interface that employs some kind of well-defined INTERFACE DESCRIPTION[1] [32]. The INTERFACE DESCRIPTION contains all interface details about the service (i.e. the operation signatures and how these operations can be accessed remotely). The service advertises itself at a central service, the lookup service. Applications can therefore look up the advertised services by name or properties to find details of how to interact with the service.

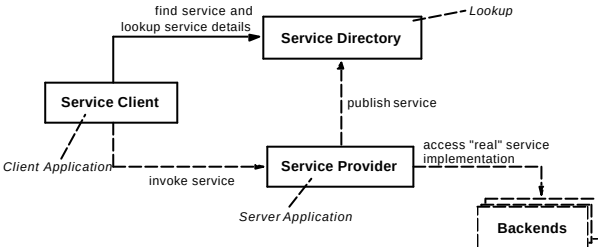

Figure 1: Lookup of services in a SOA

Figure 1 illustrates this basic architecture. A service provider offers a service to service clients. Often the service is not realized fully by the service provider implementation, but also by a number of backends, such as server applications (other SOAs or middleware-based systems such as CORBA or RMI systems), ERP systems, databases, legacy systems, and so forth. Flexible integration of heterogeneous backend systems is a central goal of a SOA. Even though the use of backend systems is of course optional, it is an important characteristic of SOAs.

A central role in this architecture plays the pattern LOOKUP [17, 32]: services are published in a service directory, and clients can lookup services. Developers usually assign logical OBJECT IDS [32] to services to identify them. Because OBJECT IDS are valid only in the context of a specific server application, however, services in different server applications might have the same OBJECT ID. An ABSOLUTE OBJECT

[1] Note that we write pattern names in SMALLCAPS font.

REFERENCE [32] solves this problem by extending OBJECT IDS to include location information, such as host name and port.

The LOOKUP pattern can be used to lookup the ABSOLUTE OBJECT REFERENCES of a service. This is done by querying for properties (e.g. provided as key/value pairs) of the service and other details about them, such as the INTERFACE DESCRIPTION of the service, a location where the INTERFACE DESCRIPTION can be retrieved (downloaded), or other metadata about the service. Note that the service client is often itself a service provider, leading to the composition of multiple services.

3. Service Contracts

Central to our understanding of services is the notion that services reflect a contract between the service provider and service clients. This view is shared by the authors of other reference models for SOAs [7, 24]. The concept derives from the design-by-contract concept [19], originally developed for software modules. In essences, service contracts define the interaction between service client and service provider. The reason for using the design-by-contract approach is that a service needs to be specified a step further than simple remote interactions, such as RPC-based invocations in a middleware. The elements of a service contract include the following information about a service:

- communication protocols
- message types, operations, operation parameters, and exceptions
- message formats, encodings, and payload protocols
- pre- and post-conditions, sequencing requirements, side-effects, etc.
- operational behavior, legal obligations, service-level agreements, etc.
- directory service

Not all of these contract elements can be expressed with today's Web services implementations easily. Communication channels and messages are usually described with INTERFACE DESCRIPTIONS. The INTERFACE DESCRIPTION of a SOA needs to be more sophisticated than the INTERFACE DESCRIPTIONS of (OO-)RPC distributed object middleware, however, because it needs to be able to describe a wide variety of message types, formats, encodings, payload, communication protocols, etc.

LOOKUP plays an important role in a SOA because it is used to locate or obtain the INTERFACE DESCRIPTION and ABSOLUTE OBJECT REFERENCE of a service. In addition, some lookup services provide more sophisticated information to describe the service information that is missing in the INTERFACE DESCRIPTION, such as operational behaviour, legal obligations, and service-level agreements. In other words, in addition to the lookup of ABSOLUTE OBJECT REFERENCES, the SOA lookup service might offer other elements of the service contract.

The SOA can also be extended with custom directories or repositories allowing for LOOKUP of domain-specific service properties or metadata of services. These might be accompanied by domain-specific schemas or ontologies, as for instance industry-specific XML schemas like OFX [22] or MISMO [21]. A service contract is usually realized by a mixture of explicit and implicit contract specifications. The above described elements are

often described as explicit service contract specifications – most often provided in electronic form. In principle all these elements can also be specified only implicitly or non-electronically.

This would be very inconvenient for the technical specification elements because it would be cumbersome, error-prone, and costly, if for instance, the ABSOLUTE OBJECT REFERENCES would not be retrieved automatically but instead distributed by hand. Hard-wiring ABSOLUTE OBJECT REFERENCES into a client is also not advisable because this would contradict the principle of loose-coupling: a service client should be relatively independent of the location where the service is executed. These technical service contract elements should therefore be specified in some explicit, electronic form, ideally accessible at runtime – for instance by using the LOOKUP pattern.

Some other service contract elements, however, are often specified only implicitly or non-electronically. Examples are the documentation of the services behaviour and its implied semantics, business agreements, quality of service (QoS) guarantees, legal obligations, etc. These elements might also be needed in electronic form, when the service client needs to monitor or verify the contract, or when the service provider needs to verify or monitor the quality of the service. For instance, the client or server might observe QoS characteristics of the service (using the pattern QOS OBSERVER [32]), or check that specific business agreements are not violated. In general, monitoring and verification can be implemented using INVOCATION INTERCEPTORS [32] or OBSERVERS [13] for the service interface and adapter (described in the next section).

4. Service interface and adapter

Often a SOA is used within larger client and server applications, and the services are just used for integration purposes. Then it is advisable to introduce a service interface to the server application and a service adapter on the client side. Both are separated from the rest of the application, and encapsulate all communication issues. This way the client and server applications are isolated from changes in the service contract or the SOA in general. Figure 2 illustrates this design.

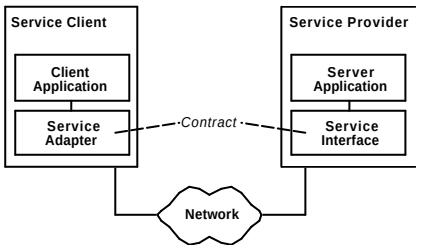

Figure 2: Service interface and adapter

Note that the service interface and adapter encapsulate service contracts described before. The service adapter can be realized using the PROXY pattern [13, 5], which generally describes how to realize a placeholder for an object or component. In this case the service adapter is a remote PROXY to the service interface, which itself wraps the server application. This wrapping architecture follows the pattern COMPONENT WRAPPER [35], which generally describes how to integrate different kinds of components.

An important task of the service interface and adapter are synchronization issues. Services are sometimes message-oriented, sometimes they are RPC-oriented. For realizing messages, sometimes reliable messaging protocols are used, sometimes unreliable asynchronous RPC is used. Both client and server applications may have to support many different service adapters and service interfaces, supporting different models. Somewhere these different ways to access services need to be synchronized, or mapped to asynchronous invocation models used in the client and server. On client side, invocation asynchrony patterns (see [32]) or messaging patterns (see [15]) can be used. Similarly, the service interface on server side must receive asynchronous messages, perform the invocation (and perhaps wait synchronously for the result), and then send a reply message to the client.

5. SOA layers

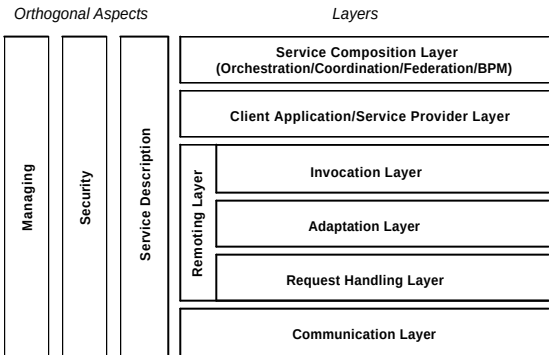

Figure 3: Client and server SOA layers

Now that we have described the overall architecture, let us take a look inside the message processing architecture of a SOA. A SOA generally has a highly symmetrical architecture on client side and server side, as it can (also) be found in many modern distributed object middleware systems. In a SOA the following LAYERS [5] can be identified (see also Figure 3):

- *Service composition.* The top-level layer of a SOA deals with the composition of services and is optional. At this layer service orchestration, service coordination, service federation, or business process management (BPM) functionalities are implemented.
- *Client application/service provider.* This layer consists of clients that perform invocations and the actual implementations of the services.
- *Remoting.* This layer implements the middleware functionalities of a SOA (for instance a Web services framework). Usually, these details of the client side and the server side are hidden in a BROKER architecture [5]: a BROKER hides and mediates all communication between the objects or components of a system. The remoting layer consists itself of three layers: invocation, adaptation, request handling. Beneath the application layer, the patterns CLIENT PROXY [32], REQUESTOR [32], and INVOKER [32] are responsible for marshaling/demarshaling and

multiplexing/demultiplexing of invocations/replies. The adaptation layer, often implemented using the pattern INVOCATION INTERCEPTOR [32], is responsible for adapting invocations and replies in the message flow. The request handling layer provides a CLIENT REQUEST HANDLER [32] and SERVER REQUEST HANDLER [32]. These two patterns are responsible for the basic tasks of establishing connections and message passing between client and server.

- *Communication.* The communication layer is responsible for defining the basic message flow and managing the operating system resources, such as connections, handles, or threads.

In addition to the basic layers that handle the message flow in a SOA, there are a number of orthogonal extension tasks that must be implemented across a number of these layers. Examples of such extensions are: management functionalities for services, security of services, and the description of services, e.g. in service contracts.

6. Adaptation in the Remoting Layer

A characteristic property of SOAs is that they are highly adaptable in the remoting layer:

- Possibly different communication protocols and styles must be supported, even at the same time.
- As depicted in Figure 3 a number of orthogonal tasks might need to be configured for service, such as management functionalities for services, security of services, monitoring of service contracts, logging, etc.
- The service might not be implemented by the service object itself, but by a backend. A heterogeneous set of backends should be supported.

In addition to these requirements, a SOA usually has to be able to be adapted at runtime. Thus a highly dynamic and flexible architecture is required that supports respective runtime variation points. Figure 4 shows the main variation points in a SOA's remoting layer, corresponding to the variation requirements. These are explained in more detail in the remainder of this section.

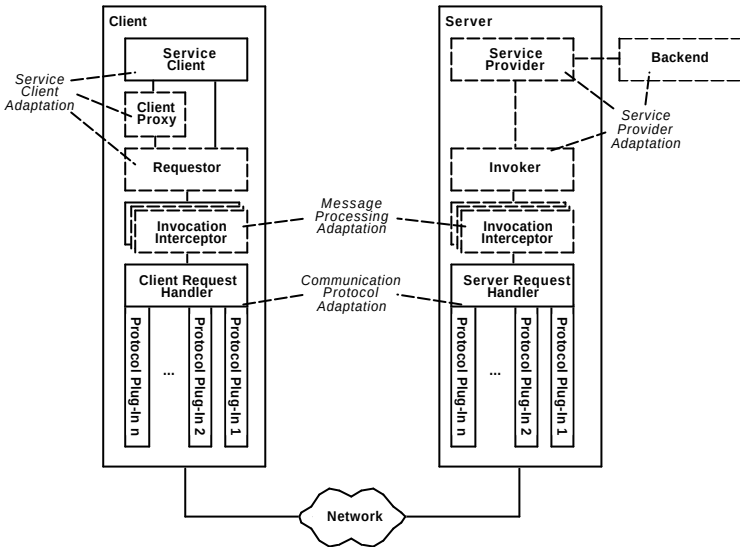

Figure 4: Main variation points in a SOA's remoting layer

6.1. Communication Protocol Adaptation

As mentioned above, on the lowest layer, the communication layer, we require a high flexibility regarding the protocols used, because usually a SOA allows for a number of communication protocols to be used. These communication protocols might require different styles of communication, such as synchronous RPC, asynchronous RPC, messaging, publish/subscribe, and others.

Variation at the communication layer is usually handled via PROTOCOL PLUG-INS [32]. PROTOCOL PLUG-INS extend the CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER with support for multiple, exchangeable communication protocols. They provide a common interface to allow them to be configured from the higher layers.

6.2. Adaptation of Message Processing

There is a distinct adaptation layer in the SOA architecture, shown in Figure 3. This adaptation layer is often realized by the INVOCATION INTERCEPTOR pattern. INVOCATION INTERCEPTORS are automatically triggered before and after request and reply messages pass the INVOKER or REQUESTOR. The interceptor intercepts the message at these spots and can add services to the invocation.

Adapting the message processing is necessary to handle various control tasks, like management and logging, or pervasive tasks, like security. These tasks need to be flexibly configurable. In addition, in a SOA, there might be multiple payload formats with different marshalling rules. Thus there should be some way to handle these flexibly as well. This is often done using custom MARSHALLERS [32] configured as INVOCATION INTERCEPTORS.

Usually, the same INVOCATION INTERCEPTOR architecture can be used on client and server side. For many tasks, we need to pass additional information between client and server. For instance, for an authentication interceptor on the server side we require

additional information to be supplied by the client side: the security credentials (such as user name and password). These can be provided by an INVOCATION INTERCEPTOR on client side. However, how to transport this information from client to server? This is the task of the pattern INVOCATION CONTEXT [32]: the INVOCATION CONTEXT bundles contextual information in an extensible data structure that is transferred between client and remote object with every remote invocation.

6.3. Service provider adaptation

The service provider is the remote object realizing the service. Often the service provider does not realize the service functionality solely, but instead uses one or more backends. When a SOA is used for integration tasks, it should support multiple backend types. The goal of providing support for service provider adaptation in a SOA is that only the service interfaces are exposed and service internals are hidden from the service client. This way it is possible to provide integration of any kind of backend with one common service provider model.

Service provider adaptation needs to be supported by the remote objects realizing the service, as well as by the INVOKER that is used for invoking them. A common realization of service provider adaptation is to provide one INVOKER type for each backend type, and make INVOKERS flexibly exchangeable (e.g. using deployment descriptors). INVOKERS used in this way realize the pattern COMPONENT WRAPPER [35], which generally describes how to wrap an external component using a first-class object of the programming language. Use of COMPONENT WRAPPERS gives the application a central, white-box access point to the component. Here, the component access can be customized without interfering with the client or the component implementation. Because all components are integrated in the same way, a variation point for white-box extension by component's clients is provided for each component in a system.

Service providers and INVOKERS need to be tightly integrated with the LIFECYCLE MANAGER [32], which provides a central place for lifecycle management in the SOA. This is because it is important that the INVOKER selects the best-suited lifecycle strategy pattern for the service. Some services might be implemented as STATIC INSTANCES [32], who live from application startup to its termination. For most systems that access a backend, however, it advisable to use PER-REQUEST INSTANCES [32], who live only as long as a single invocation. When session state needs to be maintained between invocations, CLIENT-DEPENDENT INSTANCES [32] should be used. The CLIENT DEPENDENT INSTANCE must implement a session model and a LEASING model [17] compatible with the model of the backend. The LIFECYCLE MANAGER should also handle resource management tasks, such as POOLING [17] or LAZY ACQUISITION [17].

6.4. Service client adaptation

Service clients should also be adapted, but the goal of service client adaptation is different than on the server side: here independence of service realization and loose coupling are important. As explained above, service client adaptation is mainly reached by LOOKUP of services and well-defined INTERFACE DESCRIPTIONS. Other aspects of service client adaptation are the flexible (e.g. on-the-fly) generation of CLIENT PROXIES or the direct use of REQUESTORS to construct invocations on-the-fly. Finally the client must be adapted to how the result is sent back (if there is any). Here, usually synchronous blocking, or one of the client invocation asynchrony patterns, described in [32], is used.

(these are: FIRE AND FORGET, SYNC WITH SERVER, POLL OBJECT, and RESULT CALLBACK).

7. SOA and Business Processes

If we leverage the idea of a SOA and introduce the decoupling of process control logic by a service orchestration layer, we will end up with a process-driven concept for SOA. In fact, decoupling process logic implies another level of organizational flexibility. Actually, this is the very point where the perspectives of technical architecture and organizational architecture tend to merge via the process paradigm. From a business perspective, Process Engineering aims at optimizing the business processes of an organization. It is changes to those business processes that need to be implemented quickly, in order to cope with a dynamic business environment. The latest definitions of the term Business Process Management (BPM) illustrate that workflow technology has become an important conceptual artefact that brings the formerly separate worlds of organizational and technical design into an interdependent context [23]. Conceptually, Business Process Management implies, on a technical level, the design of technological platforms that allow organizational flexibility.

The business demand of such platforms has been identified by the management sciences as well [26]. The design of these platforms is already strongly demanded by many industries, as the time to react on organizational change requirements is becoming shorter and shorter. The IT of an organization is the key enabling factor, as far as this aspect is concerned, because organizationally inflexible technology implies cost intense implementation of organizational changes. As many enterprises are shifting to process-oriented organizations, IT platforms have to consider this process approach conceptually. It can be expected that process-orientation and its effects become even more important in the future, because organizations will build flexible process-driven networks that form virtual companies via process-oriented technology [18]. For this reason, it is important to address the link between business processes and SOA.

7.1. A High-Level Pattern Perspective

At the most abstract pattern perspective, there are several important patterns that follow a process-oriented approach. Those patterns can be identified as the MANAGED COLLABORATION, MANAGED PUBLIC PROCESSES, MANAGED PUBLIC AND PRIVATE PROCESSES, and EXPOSED BUSINESS SERVICES [1]. Mapped to SOA these patterns address variations of service orchestration within an enterprise or across enterprise boundaries. However, they represent design guidelines at a high level where principle collaborative decisions are made at the business level – these patterns help on the actual decision what collaborative patterns are appropriate for a certain business problem and thus help finding appropriate patterns of service collaboration.

Concerning integration of SOA and business processes there are several important integration patterns, such as ROUTER, BROKER, and MANAGED PROCESS [1]. These are general patterns that are, in combination, suitable for bridging the two views of SOA and business processes. The following sections will elaborate on this in more detail.

7.2. Integrating Services and Processes

Fundamentally, a process-aware information system can be shaped by five perspectives: *data* (or information), *resource* (or organization), *control flow* (or process), *task* (or function), and *operation* (or application) [34, 16]. This view can be mapped to the SOA approach: services are a specialization of the general operation perspective. The process control flow orchestrates the services via different process steps, the tasks. The operations executed by tasks in a control flow correspond to service invocations. The following paragraphs will illustrate how these perspectives need to be addressed at the Service Composition Layer in a SOA.

As far as the data perspective is concerned, it is necessary to distinguish between process control data and the business objects that are transformed via the process flow. An example of such a business object could be a customer order that is being processed via a process flow. The actual processing of that order is controlled by control data that depicts the routing rules, for instance. Each process step can be interpreted as a certain state of the business object. In a SOA this means that service orchestration will also have to deal with control data and business objects being transformed and passed from one orchestration step to the next one.

The control flow perspective is captured by a process engine. Generally, today's process engines follow two possible paradigms. The traditional paradigm is a strictly structured process flow that dictates a strict ordering of activities, as implemented by engines like IBM's WebSphere MQ Workflow or Staffware. The flexibly structured paradigm is rather innovative and does not dictate a strict ordering of activities, exceptions are rather the rule. An example of that approach is the product FLOWer from Pallas Athena [3].

In order to create the link between an activity of a process and a service, integration logic is required (represented by a process flow). We classify this type of integration logic as *process integration logic*. For this reason, we distinguish between two general types of process flow: *macroflow* representing the higher-level business process, and *microflow* addressing the process flow within a macroflow activity. The distinction between micro- and macroflow is a conceptual decision in order to be able to design process steps at the right level of granularity when designing at the long running business process level (macroflow) or the short running, more technical level (microflow). This conceptual decision is thus important for separating the business problems from the more technical/application problem space.

Concerning the microflow level, the BROKER and ROUTER patterns are important in order to model communication between a process-step and services at an endpoint at a technical level. The request for service invocation sent by the process-step must be routed to the right endpoint, which is done by a BROKER.

Accordingly, in message-oriented communication between a process engine and a service, various messaging patterns like MESSAGE ROUTER, MESSAGE TRANSLATOR, and their specializations like CONTENT-BASED ROUTER, DYNAMIC ROUTER, ENVELOPE WRAPPER, CONTENT ENRICHER are important, to name just a few [15]. Those patterns are used to route requests of service invocations sent by a process-step to the right endpoint, route the corresponding responses backwards, and perform data transformation. Figure 5 shows the corresponding meta-model with the roles of services in a process-aware system.

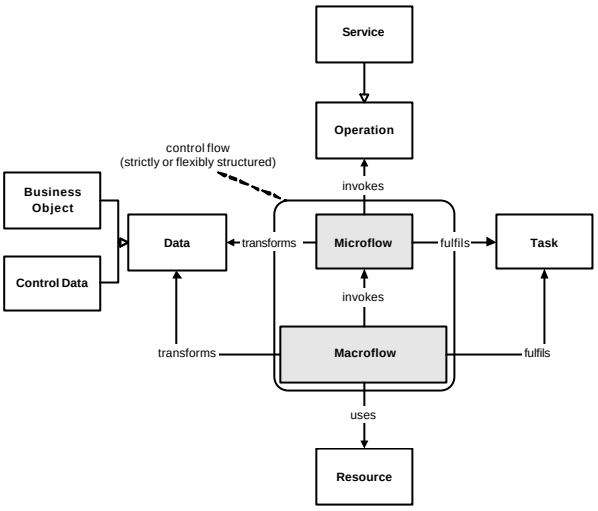

Figure 5: Meta-model showing the link between SOA and workflow processes

7.3. Process Service Levels

On the one hand a process flow orchestrates the service invocations, but on the other hand a business process may be exposed itself as a service. Thus, a process has a well defined service INTERFACE DESCRIPTION. This applies to the microflow and macroflow level.

Figure 6 correspondingly illustrates several levels of service invocation, which can be classified as follows:

- a *business process service* represents a business process being exposed as a service
- a *process integration service* depicts process integration logic at the microflow level
- a *business application service* is a service that is offering functionality of a business application

The control flow design, both at microflow and macroflow level, usually follows (some of) the workflow patterns [31, 30, 29]. These workflow patterns address business requirements related to basic control flow, workflow structure, synchronisation, branching, cancellation, and multiple instantiation.

Moreover, other control flow patterns apply that can be named as ACTIVITY INTERRUPT, PROCESS INTERRUPT TRANSITION, and PROCESS BASED ERROR MANAGEMENT [14]. Those patterns address problems that appear during process modelling when taking a broader architectural perspective. Managing errors returned by an invoked service via the process flow is addressed by the PROCESS BASED ERROR MANAGEMENT pattern. Terminating a process in a controlled way is addressed by the PROCESS INTERRUPT TRANSITION pattern, and interrupting the processing of an activity without the loss of data is addressed by the ACTIVITY INTERRUPT pattern.

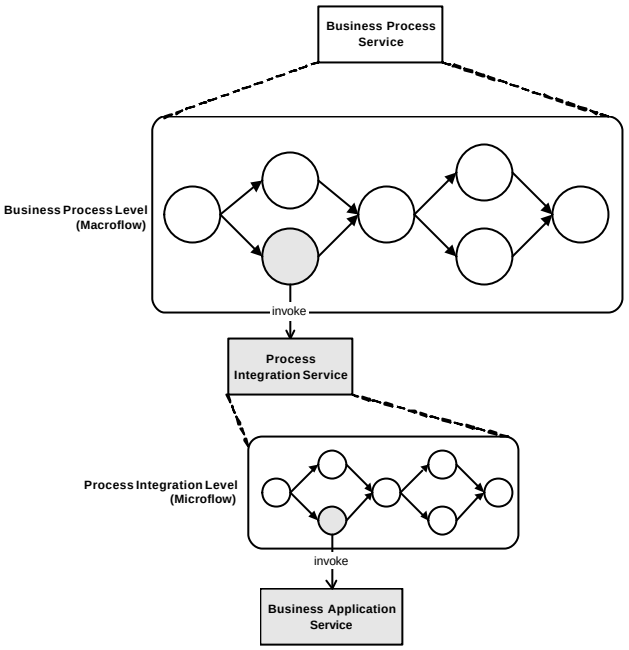

Figure 6: Levels of process service invocation

As previously mentioned, business objects are manipulated via the process steps which are represented by services. In this context the ENTITY pattern [12] is important, as those business objects actually represent entities in a REPOSITORY [12], in which the business objects depict a CANONICAL DATA MODEL [15] for storing process relevant business data. As many process engines struggle with changes to control data at runtime the GENERIC PROCESS CONTROL STRUCTURE pattern must be considered [14], which illustrates the design of a control data structure that is unlikely to change.

Furthermore, business objects can concurrently be modified by different process instances, and for this reason, BUSINESS OBJECT REFERENCES must be part of the control data [14]. Those BUSINESS OBJECT REFERENCES are pointers to business objects in a REPOSITORY and the concrete business objects can thus be accessed concurrently via these references. Again, these patterns apply to the macroflow and microflow level.

7.4. Enterprise Service Bus (ESB)

The ENTERPRISE SERVICE BUS is based on a MESSAGE BUS [15] and is an architectural pattern that integrates concepts of SOA, EAI, and workflow management. Within this architectural pattern, various components connect to a service bus via their service interfaces. In order to connect those components to the bus, service ADAPTERS [13] are necessary. The service bus handles service requests and generally represents a message-based ROUTER and/or BROKER [1]. Service requests are routed to appropriate components connected to the bus, where services are invoked. As a result, an ESB can act as a CONTENT-BASED ROUTER, MESSAGE FILTER, DYNAMIC ROUTER, AGGREGATOR, or MESSAGE BROKER to name a few message routing patterns [15].

Additionally, message transformation patterns like NORMALIZER, ENVELOPE WRAPPER, or CONTENT ENRICHER are applied by the bus in order to integrate different service interfaces. Often a REPOSITORY of business objects is connected to the service bus.

In some cases the service bus and the microflow engine are implemented by the same component, e.g. a message integration middleware like IBM's WebSphere Business Integration Message Broker or Microsoft BizTalk, for instance. That means the service bus itself implements process integration services. Within an ESB microflows and macroflows are represented as a PROCESS MANAGER [15].

Thus, the service bus is connected to the whole internal service infrastructure and all services communicate via the bus. Access to those services is classified by different service types. For this reason, it is possible to LOOKUP services by their service type, e.g. process services, information services, interaction services, partner services, etc. Figure 7 shows the ESB as an architectural pattern. For example, the ENTERPRISE SERVICE BUS pattern is implemented by IBM's Business Integration Reference Architecture consisting of products from the WebSphere family. The Service Provider Delivery Environment (SPDE) architecture is an implementation of this reference architecture for the Telecommunications industry.

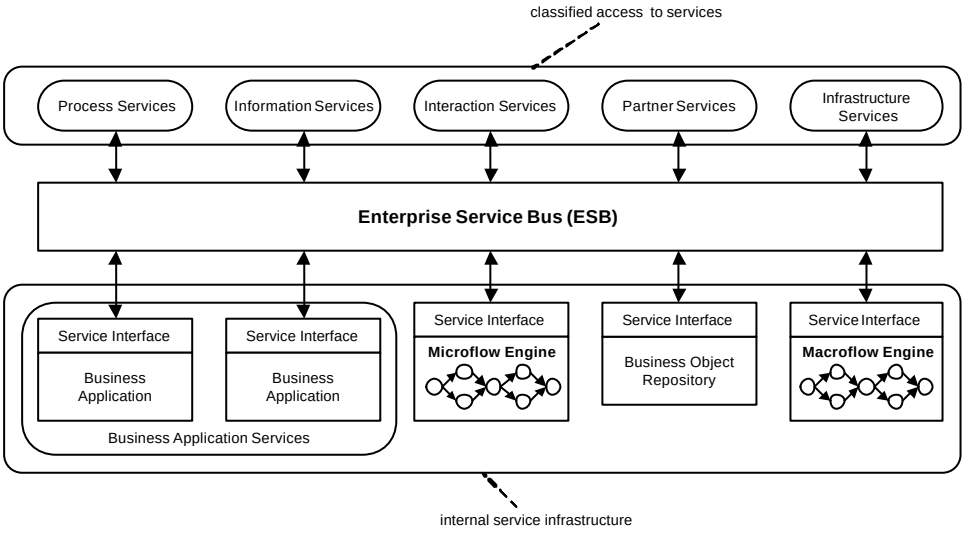

Figure 7: Enterprise Service Bus

7.5. Process Integration Services

Process integration services are the connection between a macroflow activity of a business process and a service interface in the backend. As those backend services can be developed and enhanced independently over time, they stand for themselves and are primarily not dependent on a process model. However, if a backend service is invoked by a macroflow activity, the result of the service invocation may be stored in a business object, and control data, based on the service result, containing the BUSINESS OBJECT REFERENCE [14], must be passed to the calling macroflow activity. Thus, integration

logic is required to establish the communication between the backend service and the macroflow activity; i.e. it is used for routing and data transformation. Basically, that communication is based on the previously mentioned message routing patterns. The business objects relevant to microflows and macroflows form the CANONICAL DATA MODEL for storing process relevant business data. From an architectural perspective it is necessary to have a flexible concept for process integration services that can be adapted according to changing workload.

In larger architectures there might be several process engines involved for microflows and macroflows that need to be connected. For each macroflow engine a process integration service ADAPTER [13] is required. A MESSAGE DISPATCHER [15] is responsible for distributing process integration service requests to different microflow engines, where the integration logic is executed by a PROCESS MANAGER [15] represented by the microflow engine. The PROCESS MANAGER coordinates the integration steps and invokes the business services in the backend. A REPOSITORY of process integration adapters contains all available adapters. This REPOSITORY and the dispatcher are CONFIGURABLE COMPONENTS [27], thus administration and configuration is possible during runtime. The request and responses are related to a specific macroflow activity by an ASYNCHRONOUS COMPLETION TOKEN [27] (or CORRELATION IDENTIFIER [15]). Figure 8 illustrates this process integration architecture.

Often there is only one macroflow and microflow engine. In that case the dispatcher might be superfluous. Some products like IBM WebSphere InterChange Server, for instance, already include adapters for different process engines off the shelf. Thus, such products depict the process integration adapter repository, the dispatcher, and the microflow execution services in one single component.

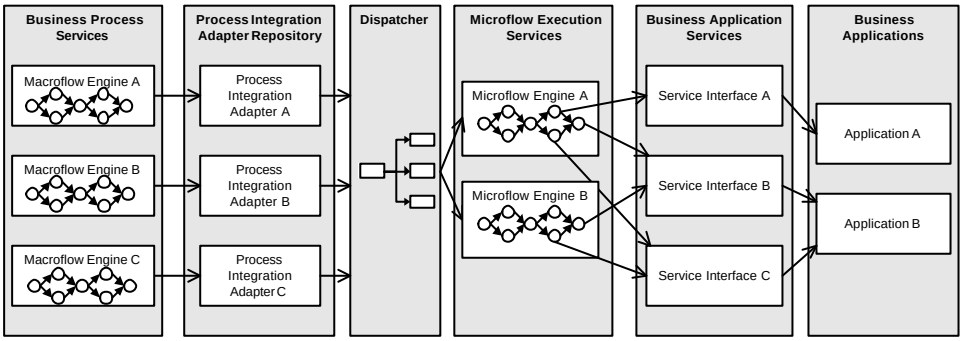

Figure 8: Process integration architecture pattern

8. Composing SOAs

In the enterprise scope, often multiple SOAs and other (distributed) systems need to be composed to work together. A simple way to reach this goal has already been discussed: we can wrap another system just like the wrapping of backends discussed before (see Figure 9). The backend does not need to be a legacy system or another non-SOA participant: the backend can be another service as well. This way, service

composition can be realized architecturally using a distributed variant of the pattern COMPONENT WRAPPER [35].

In case the server cannot be adapted, the wrapper needs to be provided in the client to adapt to an interface provided by a server.

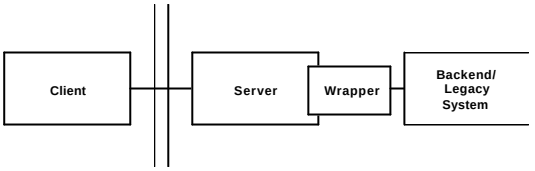

Figure 9: Backend Wrapper Style

A general alternative to a client-based or server-based backend wrapper, is a gateway. A gateway is an intermediary component, outside of client and server. It can be used to translate non-SOA invocations into SOA messages, and vice versa. It can also be used for extra tasks, such as routing, mapping RPC invocations to asynchronous messages (queuing up invocations), mapping asynchronous messages to RPC invocations (de-queuing invocations), temporarily storing messages, logging, etc.

The LOCATION FORWARDER [32] pattern is used to forward invocations to a remote object in another server application, e.g. for remote objects that the INVOKER cannot resolve locally. The LOCATION FORWARDER looks up the actual location of the remote object based on its OBJECT ID. The result of this lookup is an ABSOLUTE OBJECT REFERENCE of another remote object. The LOCATION FORWARDER has two options: either it sends the client-side distributed object middleware an update notification about the new location, so that the client can retry the invocation on the new location, or it transparently forwards the invocation to the new location. The LOCATION FORWARDER can be used as part of a SOA service to connect to other services or backends (in combination with the backend wrapper style). Alternatively, it can be used on a gateway, e.g. to realize routing or fault tolerance measures.

Sometimes a number of different frontends need to access one service. One special variant of multiple frontends is that there is more than one service offered, and each of the frontends is a different channel, such as a Web services invocation channel, Web presentations channel, CORBA channel, proprietary protocol channel, etc. If all these channels need to be served by the same services, then it will be advisable to introduce a SERVICE ABSTRACTION LAYER [33]. A SERVICE ABSTRACTION LAYER is an extra layer to the application logic tier containing the logic to receive and delegate requests. A schematic example is depicted in Figure 10.

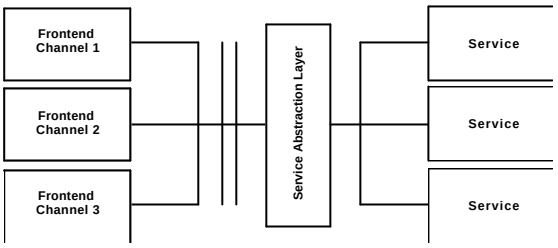

Figure 10: Service Abstraction Layer

9. Related Work

A number of authors provide technology-dependent views on SOAs. For instance, Dodani summarizes and evaluates the current best practices and technologies [9]. Because this view is highly dependent on current practices, it does not serve our goal to better understand and explain the general concepts of a SOA. Therefore, we have chosen a pattern-based approach, which concentrates more on timeless aspects than technology specifics.

Other authors provide specific surveys of composition methods in the area of Web services. Rao and Su, for instance, describe methods for automated Web service composition [25]. Dustdar and Schreiner discuss required technologies to perform service composition and composition strategies, based on currently existing composition platforms [11, 10]. These works present good overviews of the first implementations of these technologies, but again they are technology-dependent and focus only on the specific aspect of service composition. Dustdar and Schreiner also identify gaps, where essential future research work in the area of service composition is needed. Such approaches can be used as a useful supplement to our work that describes how the pattern-based concepts can be realized with today's Web services technology, and where there are still gaps that need to be closed in the future.

Many companies offer reference architectures for their platforms that are used to realize SOAs. For instance, Sun's application services reference architecture [28] presents a hardware and software platform-dependent reference architecture for Web services based SOAs. Microsoft's enterprise development reference architecture [20] provides similar architectural guidance, based on Microsoft platforms. In contrast to these architectures, we use patterns to abstract from specific platforms.

Many consulting companies offer platform-independent reference architectures for SOAs. Some are rather focusing on the technical realization with Web services and best practices (see for instance [6]). In contrast, we provide a broad view of SOA architectures, which is detailed with software patterns. Other reference architectures, such as that of the company 7irene [7], offer rather a conceptual view: here the SOA application layer and its services are seen as a conceptual bridge between the business layer and the technology layer. By using software patterns as building blocks for the reference architecture, our architecture is more detailed regarding the technical realization alternatives, and thus less abstract in its building blocks.

10. Conclusion

This paper contributes to the understanding of service-oriented architectures by mapping them to the conceptual space of patterns from various domains. The patters are successful solutions that have proven their value in numerous architectures – this is a prerequisite to qualify as a pattern according to Alexander's pattern definition [2]. Therefore, our goal was to survey and explain the "timeless" concepts in SOAs, apart from technology details. The pattern-based approach helps us not only in understanding SOAs better, but as patterns are solution guidelines, the patterns are also useful as SOA design guidelines.

In this paper, we have surveyed the essential patterns in the SOA domain. These patterns are the foundation of a pattern-based reference architecture, which combines general architectural knowledge and expertise about SOAs with specific requirements to generate particular solutions in this domain. The patterns enhance the reference architecture concept with technically detailed but yet technology-neutral solutions. In this paper, we have only informally described the cornerstones of a SOA reference architecture by informally describing the essential patterns and their relationships. As future work, we plan to further formalize the pattern relations and complete the pattern language in order to obtain a more formal model of a reference architecture. Thus, we will use the pattern survey described in this paper as a guideline for further detailed and more formal analysis following a model-driven architecture approach.

References

[1] J. Adams, S. Koushik, G. Vasuveda, and G. Calambos. Patterns for e-Business - A Strategy for Reuse. IBM Press, 2001.

[2] C. Alexander, S. Ishikawa, and M. Silverstein. The Timeless Way of Building. Oxford Univ. Press, 1979.

[3] Pallas Athena. Case Handling with FLOWer: Beyond workflow. Pallas Athena BV, Apeldoorn, The Netherlands, 2002.

[4] D. K. Barry. Web Services and Service-oriented Architectures, Morgan Kaufmann Publishers, 2003.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-oriented Software Architecture - A System of Patterns. J. Wiley and Sons Ltd., 1996.

[6] M. Champion. Towards a reference architecture for Web services. http:// www.idealliance.org/papers/dx_xml03/papers/04-01-01/04-01-01.pdf, 2004.

[7] J. Cheesman and G. Ntinolazos. The SOA reference model. http://www.7irene.com/ 7iSOA.html, 2004.

[8] J. O. Coplien. A pattern definition. http://hillside.net/patterns/definition.html, 2004.

[9] M. Dodani. Where's the SOA Beef? Journal of Object Technology, 3(10):41–46, 2004.

[10] S. Dustdar and W. Schreiner. A survey on web services composition. Technical Report TUV-1841-2004-15, Technical University of Vienna, 2004.

[11] S. Dustdar and W. Schreiner. A Survey on Web services Composition. International Journal of Web and Grid Services, 1(1), 2005.

[12] E. Evans. Domain-Driven Design - Tackling Complexity in the Heart of Software. Addison-Wesley, 2004.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[14] C. Hentrich. Six patterns for process-driven architectures. In Proceedings of the 9th Conference on Pattern Languages of Programs (EuroPLoP 2004), 2004.

[15] G. Hohpe and B. Woolf. Enterprise Integration Patterns. Addison-Wesley, 2003.

[16] S. Jablonski and C. Bussler. Workflow Management: Modeling Concepts, Architecture, and Implementation. International Thomson Computer Press, London, UK, 1996.

[17] M. Kircher and P. Jain. Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management. J. Wiley and Sons Ltd., 2004.

[18] P. McHugh. Beyond Business Process Reengineering - Towards the Holonic Enterprise. Wiley & Sons, 1995.

[19] B. Meyer. Object-Oriented Software Construction. Prentice Hall, 2nd edition, 1997.

[20] Microsoft. Enterprise development reference architecture (EDRA) version 1.0 community edition. http://www.microsoft.com/resources/practices/default.mspx, 2004.

[21] MISMO. Mortgage industry standards maintenance organization. http://www.mismo.org, 2004.

[22] OFX Consortium. Open finanical exchange. http://www.ofx.net, 2004.

[23] C. Prior. Workflow and Process Management. Maestro BPE Pty Ltd, 2003.

[24] Progess. Progress openedge release 10. http://www.progress.com/products/index.ssp, 2004.

[25] J. Rao and X. Su. A survey of automated web service composition methods. In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC 2004, San Diego, California, USA, 2004. Springer-Verlag.

[26] C. Sauer and L. Wilcocks. Establishing the Business of the Future: The Role of Organisational Architecture and Information Technologies. European Management Journal, 21, 2003.

[27] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture. J.Wiley and Sons Ltd., 2000.

[28] Sun. Sun reference architectures. http://www.sun.com/service/refarch/, 2004.

[29] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced workflow patterns. In 7th International Conference on Cooperative Information Systems (CoopIS 2000), volume 1901 of Lecture Notes in Computer Science, pages 18– 29. Springer-Verlag, Berlin, 2000.

[30] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. BETA Working Paper Series, WP 47, 2000.

[31] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow Patterns. Distributed and Parallel Databases, 14:5–51, 2003.

[32] M. Voelter, M. Kircher, and U. Zdun. Remoting Patterns. Pattern Series. John Wiley and Sons, 2004.

[33] O. Vogel. Service abstraction layer. In Proceedings of EuroPlop 2001, Irsee, Germany, July 2001.

[34] W.M.P. van der Aalst and K.M. van Hee. Workflow Management: Models, Methods, and Systems. MIT press, Cambridge, MA, 2002.

[35] U. Zdun. Some patterns of component and language integration. In Proceedings of EuroPlop 2004, Irsee, Germany, July 2004.