

Core Concepts and Use Case Scenario

Dominik Kuropka, Guido Laures, and Peter Tröger

2.1 Terminology

This chapter starts the discussion on semantic service provision by establishing a common terminology. Furthermore, we are introducing a non-trivial use case scenario, which acts as a base for the following explanations in the book. Based on both the terminology and the use case, this chapter defines the semantic service provision lifecycle based on the introduced core building blocks.

2.1.1 What is the Service?

The term service is widely adopted by the business and the software engineering community. However, both communities have a different understanding about what services are. This is usually a source for many misunderstandings in practice.

A *service oriented architecture (SOA)* [37] describes the architectural concept to organise applications and infrastructures in a business environment. Gartner defines a SOA as a technical software infrastructure which enables an interaction between service provider and service requester based on interface descriptions [139]. The granularity of a service corresponds to the granularity of the provided business function. The *Organization for the Advancement of Structured Information Standards (OASIS)* reference model defines SOA as a paradigm for the organisation and usage of distributed business capabilities [123]. Melzer et al. define SOA as the abstract concept of a software architecture which focuses on the provision, discovery and usage of services over a network [64].

Three participant roles are the crucial aspect in service oriented business: The service provider, the service broker, and the service requester respectively *service consumer* (see also Fig. 2.1).

Both terms, requester and consumer, are used to the same amount in SOA literature and standards. Therefore, both terms are used also in this book by the different authors. The SOA reference model only sticks with the service consumer term,

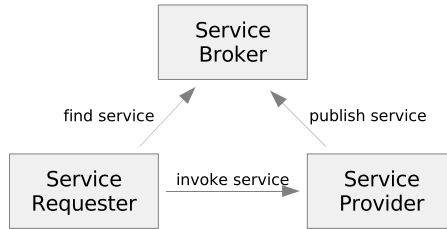


Fig. 2.1. Service oriented architecture

while other sources reason the usage of service requester with the missing exhaustion resources, which is typical in a consumption activity. Service requesters are sometimes also wrongly named as service clients, customers, or requestors.

The service requester utilizes the service through its *service interface*. A service may have several different technical interfaces. These interfaces are exposed by the service provider to give the requester access to the functionality encapsulated by the service. To properly access a service, a requester has to do specific technical things in a specific order, like sending a well structured message to a proper computer which is aligned to a predefined data format. Therefore the service providers have to publish a *service specification* to enable access to their services.

The *service broker* has the role of yellow pages, it acts as a repository for service specifications. Service providers publish their service specifications in the repository. In addition, service broker offer searching and matching tools to facilitate efficient finding of services. Common technologies to implement a service repository are *Repository for Universal Description, Discovery and Integration (UDDI)* [148], ebXML [60] and LDAP [39]. They allow a standardised storage of service specifications, natural language descriptions of the services and a semi-formal classification of the services.

In some cases, research publications also refer to the term *service oriented computing (SOC)*. The SOC approach is meanwhile widely accepted as a cross-disciplinary paradigm for distributed computing, where services act as basic building blocks for computational applications [92, 151]. Here, a service is a well-specified unit of computational work, which is again offered by a service provider. This work might be in the simplest case just the addition of two numbers; more sophisticated services offer functionalities like charging of credit cards or the initiation of a human activity for example the transportation of physical goods.

At this point, the difference between the service term in the business and the software engineering community comes to the fore. In the business community, the whole process of acquiring delivery information, physical transportation of goods and the final delivery and payment are usually seen as one (transportation) service. In SOC such an interpretation is not valid, since services have to be computational work. The physical transportation of goods as a whole therefore cannot be a service, since it can not be delivered by pure computational work. But the collection of delivery information can be a service, as long as computer based interfaces are

used. The same holds for the charging, as long as it includes some computational work like the charging of a credit card. In sum, services in SOC can only represent the computational subset or parts of real-world services in the sense of the business community. SOC could therefore be ranked as substantial part of the SOA research and paradigms.

The general crucial point of SOA approaches is the intended support of a late and dynamic service binding. The service requester should not need to bind itself at design-time to only one of the service provider. Instead, the service requester can query the broker for the most suitable services during execution time.

In theory, the resulting business competition encourages a continuous improvement of the offered services. In practice, most attempts to realise such a dynamic binding showed technical obstacles. A major challenge in finding the proper service is the derivation of service functionality from a pure syntactical specification. These descriptions only contain information about input and output data formats, but not about the service functionality and the meaning of the data structures. This is usually not a big issue with a manual integration of services at the design time of the requester application. In this case, the programmer can read additional natural language documentation or directly contact the service provider representatives. However it becomes a major obstacle with *automated late binding* of services.

A solution for this challenge is the concept of a semantic service, which is the main topic of this book. The idea behind semantic services is to extend the syntactical specification of a service by a semantic specification. This semantic specification formally describes the functionality and the meaning of the data used by the service interface. Formats and languages for a semantic specification of services are presented in Sect. 3.4, while techniques to specify existing services and to ensure a proper data integration (service grounding) are discussed in Sect. 4.3.

Another aspect also discussed in this book is the composition of services. A service composition is a partially ordered set of services in the sense of a workflow or business process [204]. It could also be understood as service based application. Today, service compositions have to be created manually which is a laborious and thus expensive task. Referring to this point semantic services also provide a benefit. With semantic services the costs for the composition of services can be reduced if assisted or automated service composition approaches are used. This topic will be discussed in detail in Chap. 5. Since service compositions offer their functionality also through a service interface, we distinguish here between a *atomic service* and a *composed service* (often also called composite service). The latter one represents the combination of different atomic services in a workflow, which itself is usable over a service interface.

For the implementation of SOA environments, many different technologies are provided by industry vendors. Web service technologies like *SOAP* and *Web Services Description Language (WSDL)* are the usual, but not the only promising technology to implement the SOA concepts. Chapter 7 discusses their technical details.

2.1.2 Service Quality

The consideration of the quality of service provisioning usually focuses on two categories of metrics: performance metrics and dependability metrics. While performance is naturally specific for the according middleware area (e.g. computational clusters vs. business service environments), dependability has a mature terminology from the history of distributed systems [16]. We therefore re-use some basic terms for dependability in our understanding of service oriented systems. Other service quality aspects beside dependability are discussed specifically in Sect. 6.3.

First, we define a correct service to deliver a functionality according to its specification, in contrast to an incorrect service, which misbehaves in comparison to its specification. A service failure is the event in time, when the delivered service functionality deviates from its specified functionality. All failures have *failure symptoms*, which might or might not be observable. This demands some understanding of a correct service behaviour, and the possibility for monitoring and rating the service behaviour to detect a failure symptom. With the existence of a service failure, the underlying system has some kind of error state. The cause of this error state in the system is called a fault.

An incorrect service shows up by a failure in the service functionality or its interface. It is caused by a system error state, which itself results from a (maybe non-observable) causing fault. It has to be noted that a service can either be a single atomic service or a composed service, consisting of several other services. The dependability concepts remain the same for both types of services.

Since the overall goal of a dependable (service oriented) system is an integrated concept [116], it consists of several sub-topics based on the above terminology. The alternation of correct and incorrect service functionality in time is used for some of the sub-concepts to describe according metrics:

- availability: The readiness of a system to provide a correct service. An according measure is the relationship between correct and incorrect service functionality over time.
- reliability: The continuity of a correct service provisioning in all cases. An according measure is the time to failure.
- safety: The absence of an catastrophic consequence on the user and the environment. As a special case, it expresses the reliability according to catastrophic failures.
- confidentiality: The absence of unauthorised information disclosure.
- integrity: The absence of improper state alteration.
- maintainability: Ability to undergo repairs and modifications. A measure is the time to restoration of the correct service after a detected failure.

Dependable system research is aware of many other dependability concepts and terms, some of them are used inflationary in the SOA research. A popular example is the notion of security as quality metric, which expresses the intend to avoid unauthorised access to service functionality. In terms of the generic dependability concepts, security represents a combination of availability for authorised users only,

confidentiality regarding unauthorised users, and integrity in terms of unauthorised usage. There are four major technologies for achieving dependable service functionality:

- fault prevention prevents the introduction or occurrence of faults.
- fault tolerance delivers a correct service even in the presence of faults, by combining error detection with system recovery.
- fault removal reduces the number or severity of faults.
- fault forecasting estimates the present or future number of faults, and their incidence.

system recovery denotes the movement from a faulty system state to a correct system state. The recovery from faults can be performed by error handling (roll-back, compensation, roll-forward) or fault handling (diagnosis, isolation, reconfiguration, re-initialization).

In SOA environments, dependability approaches usually concentrate on service compositions as unit of dependability. The system term here can be mapped to the different atomic services forming the composition. For fault tolerance, error handling works on the system (= service composition) state during runtime, while the fault handling concentrates on identifying and isolating a faulty atomic service.

In this book, we will see both error handling approaches: fault handling for service compositions, as well as fault handling strategies for atomic services. Fault removal is accompanied by manual tasks during development time (verification, validation, fault injection) or runtime (corrective maintenance, preventive maintenance), and will therefore not be in focus in this book. Also fault prevention is mainly attained during development time. Fault forecasting maps to the concept of service profiling, where the analysis of historical service behaviour is used to compute and consider dependability measures for a service.

2.1.3 Service-Level Agreements

The term of a *Service Level Agreement (SLA)* originates from an industrial context. It is frequently (mis-)used by research and industry people in SOA discussions and publications. Different communities, such as networking, multimedia, grid computing, or agent technology people, use the SLA-term in their own environment in different manners.

In networking environments, data centres and independent software vendors have several standards and technologies to deal with performance objectives for network connections [142], such as data rate, packet delay, latency, error rate, or network up-time. The multimedia community defined quality models for adaptive multichannel information systems, which are implemented by resource partitioning techniques. The Grid computing community sees SLA simply as formulation of resource requirements [12], while the agent community sees SLA as subject of negotiation between different software agents, in order to support full-automated agreement creation.

Due to this widespread, but unspecified application of SLA terminology, we try to give here common base for the SLA discussions in this book. An *agreement* in general can be seen as the concordant declaration of some intention. It includes at least some kind of statement about the agreement, and the exchange of contractual documents. In real world, these contractual agreements can regulate purchases, leasing activities, a permanent work relationship—or some kind of service.

When we interpret SOA as the mapping of business environments and conditions to the world of software, *service-level* agreements define the conditions under which a business service is provided to a consumer. This can relate to performance, cost, reliability, security, or other issues relevant for the service consumer. Obligations and provided functionalities are defined for both partners, but also penalties if the negotiated properties are not fulfilled [120].

The service requester has specific requirements and expectations on the service. The level of fulfilment for these requirements relates to the costs of the service—cheaper services have a lower level of expectable service quality metrics. The service provider has the task of fulfilling all negotiated contracts, while maximising the utilisation of the resources and minimising the costs for contract penalties. Typical service-level agreements contain quality guarantees in different granularity levels, which potentially have a stochastic definition:

The maximum response time for 95 percent of the requests should be below 2 seconds.

Technical properties of such a specification are usually named as service-level specification or service-level objective. The promised behaviour is formulated as logical expression, which is checked through measured values during runtime. Beside these technical aspects, also organisational requirements (e.g. violation notification) or commercial aspects (e.g. price for SLA violation) can be a part of a SLA definition.

2.2 Use Case Scenario

This section introduces a use case common to all chapters of this book. Concepts and technologies for semantic service provision will be illustrated by referring to this use case. It is settled in the hosting services industry. However, it is important to keep in mind that this is just one specific example. Semantic service provision and the concepts described in this book are independent from industrial domains.

2.2.1 Introduction

The use case presented in this section bases on the *business-to-business (B2B)* wholesale model of an Austrian *Internet Service Provider (ISP)* called Hostit. It specialises on products like domain registration, web hosting solutions and messaging services, but not on providing the Internet access itself.

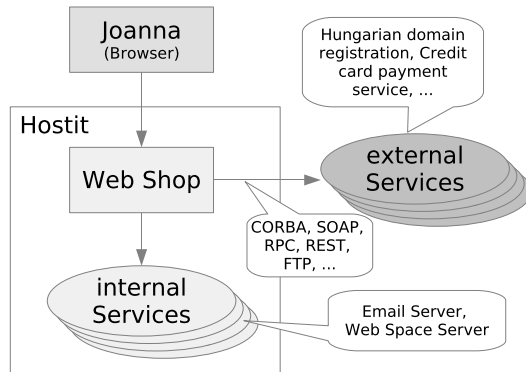


Fig. 2.2. The available B2C solution

Let us consider the *Hostit business-to-consumer (B2C)* hosting product shop solution shown in Fig. 2.2. *Joanna*, an Hungarian woman, wants to become a *Hostit* customer. She uses her browser to access the *Hostit* web shop and order the hosting solution she desires. She is interested in registering her own domain (*joanna.hu*) with e-mail addresses and web space. She wants to use her credit card to pay for the products. Using the *Hostit* web shop she is able to specify all configuration items needed for the order processing like the intended domain name, the payment method, and so on. To process an order the *Hostit* web shop application accesses the needed company-internal services as well as services from external partners. The service selection highly depends on the order configuration, respectively the service invocation order is determined by *Hostit*'s business processes.

To process *Joanna*'s order *Hostit* has to make use of external partner services for domain registration, operating and maintaining of *Domain Name Server (DNS)* information, web hosting configuration, and payment. Which services to use highly depends on the customer's demands. For instance, the service used for domain registration depends on the customer's desired top-level domain. Assignment of domains with *.com* or *.org* endings are governed by *Internet Corporation for Assigned Names and Numbers (ICANN)* while for example national domains like *.de* or *.at* are assigned by *Deutsches Network Information Center (DENIC)* in Germany or *nic.at* in Austria. In *Joanna*'s case, the domain registration service to use is *Network Information Centre (NIC)* Hungary. Internet service providers accredited by the supervising registrars can register domains by using dedicated interfaces of the registrars domain database. Web hosting services of internet providers encapsulate interfaces for web hosting systems. They allow allocation of web space to users while enforcing fine-grained restrictions on data volume, traffic and e-mail configuration.

Hostit is interested in expanding into the German market. Thus, its cooperates with a German newspaper named *Heidelberger Zeitung*. This company wants to bundle a newspaper subscription with a web hosting and e-mail product. The idea is that subscribers automatically get their own e-mail address and domain for a one year subscription as a free add-on. To enable *Heidelberger*

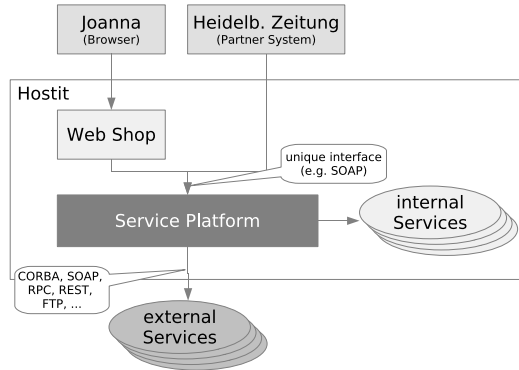


Fig. 2.3. Extension for B2B scenarios

Zeitung to order Hostit products without having to manually order them using the web client, Hostit expands its existing end-customer-centric application to a more flexible platform that can be used through various front-end solutions operated by resellers. The goal is to develop a B2B solution that reuses already available elementary business capabilities. This service reuse reduces customer acquisition and support costs for Hostit. The underlying internet service provision system requires extensions to support a more generalised B2B approach instead of a restricted B2C application. Currently, the complexity of interweaved subtasks of product ordering in the web shop and provisioning of these products through a back-end system hinders the reuse of provisioning capabilities through varying order processes.

Figure 2.3 illustrates Hostit's system architecture that supports both, the B2C and the B2B access using a central service platform. The benefit of unifying and channelling the access to the company-internal and -external services is an increased flexibility and maintainability. To integrate a new service, it is published in the service platform. Access control rights can now be centrally managed in the service platform.

2.2.2 Drawbacks in State-of-the-art Service Provision

Using a state-of-the-art service provision platform has a number of drawbacks. To integrate a new service not only the service needs to be published in the platform but also the platform clients need to be changed to benefit from the new service. This is the case because the clients of the service provision platform explicitly invoke specific services and hence will not notice newly available services on the platform automatically. Another drawback is that the service invocation order is hard-coded in the clients. Thus, clients need to be changed to implement new or adapted business processes that might be useful because of the newly available services. Last but not least a state-of-the-art service provision platform does not provide fall-back solutions in case of a service invocation error. Thus, if the invocation of a specific service fails

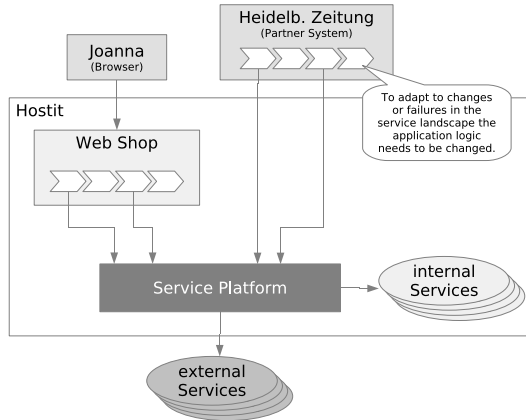


Fig. 2.4. State-of-the-art service provisioning

for instance because of a network problem on the provider side, the platform does not know if there might be another service available with an equivalent functionality.

Figure 2.4 illustrates the service invocation logic inside the web shop and the B2B systems. The little white shapes inside the web shop and the Heidelberg *Zeitung* application symbolise the steps that form the internal application logic. Some steps invoke company-internal or partner services on the service platform. The binding of steps to services is hard coded inside the application. Thus, the application logic will not make use of newly available service unless it is changed. The steps of the application logic are duplicated in the web shop and the Heidelberg *Zeitung* partner application. Thus, all these applications need to be adapted if they want to benefit from changes in the service landscape. If a service in the platform fails the application logic needs to cope with this failure. Thus, it needs to know if there are functionally equivalent services available in the landscape and how they are invoked.

2.2.3 Semantic Services

Semantic services promise to overcome the drawbacks of traditional service provision approaches. Therefore, *Hostit* engages *Steve*, an expert in semantic services, to plan a *Hostit* service provision architecture that leverage innovative technologies and methodologies from that area. Steve's first draft of a new *Hostit* architecture looks much alike the current solution. However, there are a few changes that aim at an improved flexibility, adaptability and maintainability of the entire architecture. Figure 2.5 shows Steve's first high-level plan for the semantic service solution.

One of the main differences compared to the architecture shown in Fig. 2.4 is the reduced complexity of the clients of the *Hostit* service platform. The clients now access the service by providing semantic service requests. They do not have to state explicitly which services to use but describe their needs using semantics and

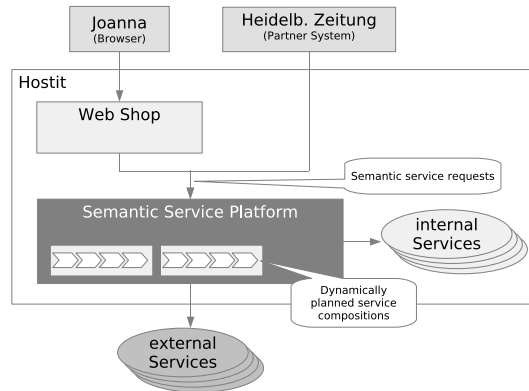


Fig. 2.5. Draft architecture with semantic service provisioning

let the platform find out which of the available services to use for the fulfilment of this request. Thus, the logic of composed services that together implement an order processing is now in the platform, not in the client. Even more important, the logic is not hard coded but it is composed at runtime and thus, capable of dealing with changes in the service landscape. As soon as a new service is published in the platform, it is considered as a candidate to be used in a service composition. Furthermore, if a service fails the platform is able to identify semantically equivalent services or service compositions and use these instead.

The components of such a semantic service provisioning platform, how it uses semantics for dynamic service compositions and which technologies are available for building up such a platform are the topics of this book. Thus, having read this book you will understand how *Hostit* can set up a platform that offers all the benefits mentioned above.

2.3 Adaptive Service Provision Approach

The *traditional approach* for using (Web) services in SOA bases on static binding of services. During design-time of the application the developer selects proper services and binds them to the applications. This implicates that the applications can only adapt in a very limited manner to changes in the service environment. For intra-organizational scenarios this deficiency is often tolerable, since the service environment is under full control of one organisation and unplanned changes are rather infrequent. In contrast to this, static binding of services is problematic when it comes to inter-organisational scenarios where major parts of the service environment are not under control of one entity. Static binding of services suffer from the following drawbacks:

- *Poor utilisation of new services:* Static service binding enforces a manual adaptation of existing applications to include new, cheaper or better services. This

causes additional costs in the maintenance of applications and it limits the timeliness of adaptations as well as the ease of implementation of new business models which usually are combined with new services.

- *Poor reliability*: In inter-organisational scenarios services may fail or disappear at any time for a large variety of reasons: communication and network failures, overload, internal failures or simply disappearance of the providing organisation. In all these cases, the static binding of services causes a (partly or even total) failure of the depending applications. In situations where it is impossible to simply sit such a failure out (e.g. because of running costs) a time consuming and costly manual adaptation of the applications is needed.

In this book we propose an approach which avoids the above mentioned drawbacks by enabling a dynamic binding of services which leads to an *adaptive service provisioning*. Instead of simply binding services to applications at design-time we propose a sophisticated and adaptive service delivery life-cycle as shown in Fig. 2.6. The entry or initial point of this delivery life-cycle is a semantic service request. In contrast to a static service binding the semantic service request does consist of a description of what shall be achieved and not which concrete service has to be executed. The semantic service request describes the initial and goal state and consist therefore among other things out of the given data, data types, and conditions which are met by the data as well as the desired type of data and desired effects beyond. The data types, conditions and, effects are all specified in relation to a common set of concepts—the domain ontology. Referring to the book’s use case scenario from Sect. 2.2.1 a semantic service request (here formulated in natural language) might be:

Given the domain `joanna.hu` (initial state) check if this domain is available (goal state).

a more complex semantic service request might look like this:

Given the domain `joanna.hu` and the credit card with the number 1234 5678 9999 0000, registered for Joanna, expiry date 1st of January 2010 (initial state) register the domain and charge the credit card with 19.99 €.

A detailed explanation of the formalisation, its structure, and the semantic evaluation of a semantic request is presented in Chap. 5. Whereas ontologies are discussed in Chap. 3.

Now we want to give a brief overview on the various steps and cycles shown in Fig. 2.6. The *planning sub-cycle* is the first step in processing of the semantic service request. At the beginning the semantic service provision platform tries to find a service, which perfectly matches the semantic service request. Perfectly matching means that the service is able to process the given data as input, that all pre-conditions for the execution of the service are fulfilled and that the service output fits to the desired type of data and effects. In case of successful matchmaking, the Planning Sub-Cycle is completed. Otherwise the platform tries to find an abstract composition of services, which is able to meet the semantic service request. Abstract composition means that the composition does not directly bind to services.

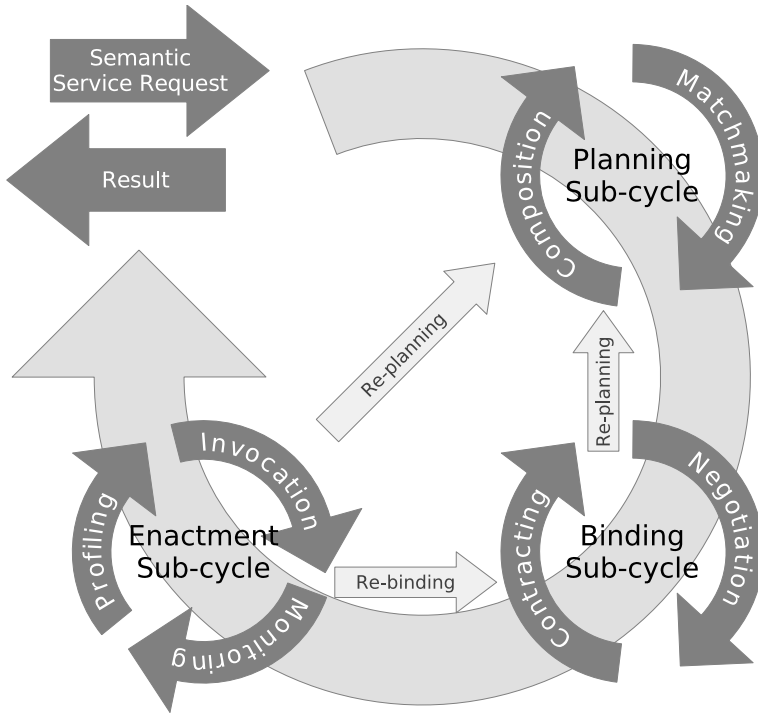


Fig. 2.6. The service delivery life-cycle

Rather the services are represented by semantic service specifications which act as place holders for the real services. This proceeding allows a late binding of services and features a better re-usability of service compositions which is useful for performance issues (e.g. by caching of compositions). The composer starts composing from the initial state of the service request. Given this state the composer searches for services, which are executable in this state. By taking their results and effects into account, the composer searches successively for executable services until either the goal state of the semantic service request is met or a further composition is not possible or reasonable. In case of successful composition or a perfect matching of an individual service, an abstract service composition representation is created and forwarded to the next sub-cycle. Otherwise the processing of the semantic service request is aborted. Details on service matchmaking are provided in Sect. 3.5 and on composition in Sect. 5.4.

Abstract service compositions are transformed into enactable service compositions by the *binding sub-cycle*. This happens by binding the semantic service specifications—which act as place holders—to concrete services. For each place holder the platform starts a negotiation on negotiable service properties with all matching services. The platform tries to find a combination of services which fit as good as possible to the desired properties of the semantic service request. It is worth men-

tioning that the properties of services which are discussed now are not the semantic functionality of the service like the above mentioned input, pre-conditions, output and effects of a service. Rather the properties describe qualities of the service execution like for example the duration or the costs per execution. These properties are limited insofar that they are not allowed to depend on the concrete input data. When an agreement with a particular service is achieved a digital contract it is set up and signed by both parties. Details on service binding including negotiation and contracting are presented in Sect. 5.5. Services do not necessarily need to support negotiation mechanisms. In such cases, the platform simply selects services by their given service properties without conducting a negotiation procedure.

The third step in processing of semantic service requests is the *enactment sub-cycle*. It receives enactable service compositions from the Binding Sub-Cycle and enacts them by successively or—where possible—invoking the scheduled services in parallel. The invocation of services is monitored by the platform for two reasons: On the one hand it is monitored to verify if the previously contracted peculiarities of properties are met. On the other hand the monitoring data is passed to a profiler which aggregates the data to a service-specific profile. This profile contains an aggregate of experiences made during past invocations of a service. It contains information like the average service execution time or reliability in the sense of observed probability of failure. This profile information can be used by the Planning and Agreement Sub-Cycles to avoid unreliable services. Further details on invocation, monitoring and profiling of services are introduced in Chap. 6. After successful enactment of the service composition the result is collected and it is send back to requester.

As already mentioned in the beginning of this section, service environments are quite dynamic when inter-organisational scenarios come into play. The service delivery life-cycle as shown in Fig. 2.6 on page 16 includes two mechanisms to handle the dynamics when it comes to explicitly considered or unconsidered failures of services: re-binding and re-planning. The definition of the term failure which is used here has already been presented in Sect. 2.1 and is compliant to the definition in [16].

Considered failures are well known failures which might occur during the execution of a service and are therefore explicitly specified as possible (even though not desired and therefore hopefully seldom) results of a service. One example for such an considered failure is the rejection of a credit card by a credit card charging service, which might occur if the credit card data is invalid or expired. Another example is the physical loss of a package by a package shipping service (for example by accident). Considered failures are handled in our platform by conducting *re-planning*. In case a considered failure occurs during the invocation of a service, the Planning Sub-Cycle is triggered to find a new composition. The composer tries to find a solution which takes the already achieved results into account including the specified undesired results of the failure. The new composition is required to reach the goal specified in the original semantic service request notwithstanding the occurred failure. (Refer to Sect. 5.4) Referencing the package shipping service this might mean that a new package is seized and send. Naturally not all considered failures can be handled by re-composition. In case of the credit card charging service a

recovery is not possible and it is even not useful if a credit card is invalid. Someone may ask why we propose to handle considered failures by re-composition instead of simply handling them by considering them directly in the original composition as it is usual in common workflow applications. There are several reasons for using our approach:

- The limitation of the composing algorithm to an optimistic composition reduces the composition time. Optimistic composition means that all specified considered failures are ignored as possible results during the composition. This reduces the number of path through a composition and thus limits the size, complexity, and search space of a composition.
- The proceeding of optimistic composition avoids infinite loops to handle failure cascades. In the package shipping service example the recovery of the package loss failure is simply done by resending a new package. However, the resent package might get lost as well. This new failure has to be recovered once again and so on.
- The flexibility is higher when compositions are re-planned at the time of failure instead of predefining static recovery mechanisms. Referring to the package shipment example, a different shipment strategy or simply a different service of a different provider might be chosen by the Planning Sub-Cycle or the following Binding Sub-Cycle if the first or second shipment fails.

In contrast to the considered, are the *unconsidered failures* not explicitly specified for a service. Unconsidered failures are usually low-level issues like network failures which are raised before or during the invocation of a service. The platform has no detailed information about semantic effects of such failures except that these failures just happen at a given point in time. For this reason it assumes in such cases of failure, that the according service simply has not been executed and thus its desired results and effects are not achieved. Such unconsidered failures are handled in up to two phases. In the first phase the platform tries to recover the failure by re-binding. This triggers a new pass of the Binding Sub-Cycle. Here an search for an alternative equivalent service to the already invoked and failed service is conducted by negotiation with proper services. If the search is successful, the new service is invoked in substitution for the old one. Else the second phase is conducted. In this second phase the platform tries to recover the unconsidered failure by re-planning the composition in the Planning Sub-Cycle. Similar to the handling of considered failures, a search for a new solution with different services is conducted which nevertheless reaches the specified goal. In case this search is successful, the Binding Sub-Cycle is invoked. After the outstanding service place holders have been bound to concrete services the new composition is enacted in the Enactment Sub-Cycle.

As a summary of this section we should adhere, that the challenges on adaptiveness and reliability of a global and thus dynamic service environment can be met by a three-staged semantic-enabled adaptive provision of services as proposed here. The next chapter will go into more details regarding the semantic aspects of service specification and service matchmaking.

Semantic Service Provisioning

Kuropka, D.; Tröger, P.; Staab, S.; Weske, M. (Eds.)

2008, XII, 226 p. 71 illus., Hardcover

ISBN: 978-3-540-78616-0