ASG—Techniques of Adaptivity*

Harald Meyer and Dominik Kuropka and Peter Tröger

Hasso-Plattner-Institute for IT-Systems-Engineering at the University of Potsdam
Prof.-Dr.-Helmert-Strasse 2-3, 14482 Potsdam, Germany
{harald.meyer|dominik.kuropka|peter.troeger}@hpi.uni-potsdam.de

Abstract. The introduction of service-orientation leads to significant improvements regarding flexibility in the choice of business partners and IT-systems. This requires an increased adaptability of enterprise software landscapes as the environment is more dynamic than the ones in traditional approaches. In this paper we present different types of adaptation scenarios for service compositions and their implementation in a service provision platform. Based on experiences from the Adaptive Services Grid (ASG) project, we show how dynamic adaptation strategies are able to support an automated selection, composition and binding of services during run-time.

Keywords: adaptive service provision, service selection, automated service composition, service recovery

1 Introduction

With the proliferation of service-enabled business, building on top of service-oriented architectures, the adaptivity of business processes, implemented by service compositions, attracts more and more attention. Adaptivity allows for process instances to deviate from the standard behavior in order to better serve the customer or handle errors and unforeseen events. In service-oriented approaches this becomes increasingly important as the environment is more dynamic than the one in traditional approaches. Services ease the integration of internal and external functionalities towards new value chains by composing services to new functionalities. However, this also means that the interdependencies between partners increase. This can have negative effects on the implemented value chains, since having dependencies to many partners increases the probability for a failure of the whole system. Furthermore, the recovery from failures—especially if external partners are involved—is a complex task. A possible solution is to have multiple partners providing equivalent services to select a different one if the current service is not working. Using manual composition and selection, this approach leads to increased development and maintenance

* This paper presents results of the Adaptive Services Grid (ASG) project (contract number 004617, call identifier FP6-2003-IST-2) funded by the Sixth Framework Program of the European Commission.

Dagstuhl Seminar Proceedings 07061
Autonomous and Adaptive Web Services
http://drops.dagstuhl.de/opus/volltexte/2007/1036

costs. In this paper, we present adaptation scenarios (e.g. unavailable services) and respective solution strategies (e.g. late-binding of selected services) realized in the Adaptive Services Grid[1] (ASG) project.

The goal of the ASG project was to develop an adaptive service provision platform, which supports automated on-demand selection and composition of services and the automated recovery in case of failures. ASG was an integrated project funded by the European Commission as a part of the 6th Framework Programme. It consisted of 24 partners from six European countries and Australia. The project started in September 2004 and was successfully finished in February 2007.

In the remaining part of this section, we will give an overview of related work and presented the schema used to describe the solution strategies. In the sections 2 to 6 we then present the adaptation scenarios the ASG platform can handle. Subsequently, we will also introduce five solution strategies. The adaptation scenarios and their solution strategies were motivated by use case scenarios provided by industry partners of the ASG project. In Section 7 we will present the two scenarios that were implemented using the platform to show the adaptation scenarios and solution strategies map to them. How all the features are put together is shown in Section 8. The paper closes with a summary in Section 9.

1.1 Related Work

The Web Service Execution Environment (WSMX) [1] is a software system that enables the creation and execution of se mantic Web services based on the Web Service Modeling Ontology (WSMO). Providers use WSMX to register and offer their services, while requesters dynamically discover, mediate and invoke Web services. In contrast to ASG, the current WSMX does not support automated service composition. But it implements late binding as a strategy for adaptation. It uses its own matchmaking component for the discovery of appropriate services. Early versions of WSMX used a string-based matchmaking mechanism for this task. Support for manual composition of services is currently under consideration.

A prominent example for adaptivity in service-oriented architectures is the Meteor-S project [2]. Meteor-S is a framework for semantic Web services supporting semantic annotation, discovery, and composition of services. It allows for the manual composition of services using semantic operation templates. These templates describe the functionality of an operation at each step of the composition. Meteor-S supports a late binding mechanism to discover and bind proper services at run-time for each operation of the composition. Non-functional service properties and composition constraints are taken into account by the implemented discovery and binding mechanisms. Furthermore, the framework supports rebinding of individual services at run-time to solve service failures. However,

[1] http://asg-platform.org

automated composition of services and negotiation of non-functional properties is not supported.

1.2 Solution Strategies

Within the following sections, we will describe several challenging situations in a service-oriented environment, where adaptation can provide the only, or at least a better than usual solution for the particular issue. All presented strategies foster adaptation as a primary aspect. Even though some of the strategies are applicable in different situations, we will describe each solution strategy only once, according to the following schema:

- *Summary*: Describes the scenario briefly.
- *Applicability*: Describes the situation in which this pattern can be used.
- *Consequences*: Lists the implications and side-effects.
- *Implementation Sketch*: Shows how the solution strategy is realized in ASG.

2 Changing service landscape

A service landscape is the set of services that makes up the available functionality of the whole service-oriented system. Typical changes to a service landscape are the addition of new services, the removal of existing services, and changes to the functionality or interface of existing services. The two latter problems are most critical ones. If a service gets removed or changed and it is already used in service compositions, the composition will no longer function. If the modification to the service landscape is just the addition of a new service, all existing service compositions will continue to function as before. Anyway, the new service might allow simpler service compositions (replacing several existing services) or a better service composition regarding its non-functional properties.

An example from one of the ASG use case scenarios is a dynamic supply chain for web hosting products. A set of services for domain availability checking, credit card authorization and charging, domain registration, and storage reservation is combined to a service composition. This composition is represented by a customer-friendly front-end, which supports the composition enactment through the acquisition of relevant input data. Due to fluctuating contracts with business partners in the IT hosting world, several service landscape changes might happen:

- A cheaper payment partner is contracted, which leads to the availability of a new service with similar functionality, but different non-functional properties.
- The domain check service for ".de" domains breaks during runtime, but another (slower) domain check service for all top-level domains is still available.
- The service for combined credit card checking and payment breaks or gets too slow, but two separate services for both functionalities from other providers are available.

In existing service provision platforms, the adaptation to such a changing service landscape needs to be performed manually. This involves the following steps:

- *Detect service landscape change*
- *Determine affected service composition*
- *Modify service compositions*

The required steps to perform manual adaptation are quite cumbersome. Especially, determining affected service compositions can be tricky. Nevertheless, automating adaptation to service landscape changes is only an option if these changes are frequent or if adaptation should not yield extensive downtime. Both of these conditions hold in service-oriented applications

A first idea for an automated solution might be to automate the process described above. This means detecting service landscape changes, determining affected service compositions, and modifying these compositions would be automated. However, this is not the approach used in ASG since this approach does not exploit the full potential of automation. Automation was also needed in ASG to deal with goal variations. Goal variations will be described in more detail in the next section. But in principle, goal variations mean that customers often have slightly different goals they want to attain. To cope with this, individual processes for each customer are required. The solution strategy to adapt here is *Automated Service Composition*. This solution strategy, together with another solution strategy, called *Late Binding* can actually be used to deal with both goal variations and changing service landscapes.

2.1 Solution Strategy: Late Binding

Summary: Using abstract service compositions and binding them to concrete services prior to enactment is a common approach in service-oriented architectures. Instead of describing the exact service that should be executed, only the type of the service is described.

Applicability: Late binding allows to adapt to changes of the service landscape. It can be used when these changes are frequent. An additional requirement is that changes can be managed by replacing just one service. Late binding does not allow for changes to the process structure. If for example, the landscape change is the deletion of a service, late binding allows us to use other services providing the same functionality. If no such service is available, but the service can be replaced by the composition of two existing services, this cannot be performed with late binding. The same applies to newly added services. The new service can be used to replace one service with the same functionality, but it cannot be used to replace several services in the same composition even though it provides their combined functionality.

Another requirement is the knowledge about equivalent services. In the simplest case, equivalent services implement the same interface. But this is clearly

a limited approach, since different service providers usually do not implement the same interfaces for the same functionality. Extended approaches model service equivalence explicitly through service types or implicitly through semantic service specifications.

Consequences: Service compositions need to be abstract. This has two implications: we need a format to specify abstract service compositions, and we need to model the according services abstractly. As no standardized format to model abstract service compositions[2] exists today, an own format needs to be implemented or an existing service composition format must be adapted. The second implication is quite interesting. If service compositions are to be modeled abstractly, how can we do that? More precisely: How do we choose the right granularity for the used services? If we model the composition too fine-granular or too coarse-granular, we might be unable to find matching services. Therefore, we need to know how the service landscape looks like and whether a required functionality is realized in one or more than one service. As this can be difficult in large (and changing) service landscapes, another approach is more feasible: Service compositions are modeled using concrete services and only abstracted afterwards, in order to allow for different bindings of the used services. This approach results in a tighter coupling of the modeled composition to the current service landscape, which leads to an reduced adaptability if the landscape changes.

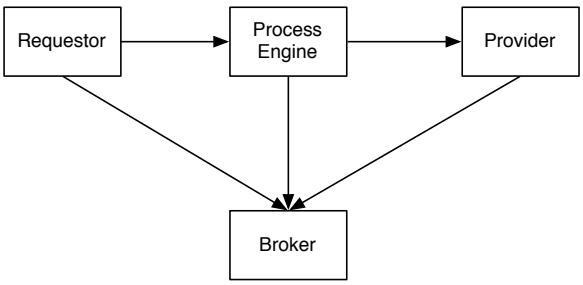


Fig. 1. Process-oriented SOA

Using late binding has also architectural consequences in a process-oriented SOA (see Figure 1). If late binding is used, the process engine is no longer only responsible for enacting compositions, but it has to perform binding prior to enactment. As we will see in the implementation sketch, the required extensions can be substantial.

Finally, another consequence of using late binding to adapt to a changing service landscape is that adaptation is not always possible. In the simplest case, if a service is removed and no equivalent service exists, service compositions

[2] WS-BPEL allows for business protocols but practice showed they are rather ill-suited for our requirements.

using the service will no longer work. More interesting is another problem: What happens to service compositions that are running during the removal? Service landscape changes after the binding was performed cannot be resolved using late binding. The composition might therefore no longer work (if a selected service is deleted or changed) or work optimal (if a better service was added). Late binding therefore only reduces the frequency of such problems. If services in compositions are bound very early, several weeks, months, or even years can be between the selection of the service and its usage. With late binding, the time span is reduced to seconds or minutes. If the service landscape changes after late binding time, another solution strategy called re-binding (Section 4.1) can be used.

Implementation Sketch: For the implementation of late service binding in a service composition, three decisions have to be made:

- How is service equivalence determined?
- How do abstract service compositions look like?
- How does the architecture of the solution looks like?

In ASG, we used semantic service specifications to determine service equivalence. Instead of referencing concrete service endpoints or WSDL descriptions, semantic preconditions and effects are specified for all services. Service discovery can used these descriptions to find suitable services. To model the resulting abstract service compositions, we extended WS-BPEL to allow for *semantic web service activities*. These do not reference a specific WSDL interface, but refer to the expected functionality by a semantic description. We decided to integrate the semantic specification based on WSML [3] directly into the WS-BPEL process. Another approach would be to reference for example semantic templates.

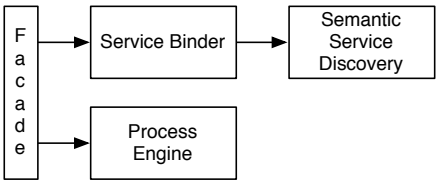


Fig. 2. Extended Process Engine for Late Binding

Finally, the usage of semantic service specifications leads to some architectural requirements. Figure 2 shows how the process engine needs to look like to realize the ASG approach of late binding. Before the composition can be enacted, the *Service Binder* needs to bind the services in the abstract service composition to concrete services. In ASG, it uses the semantic service discovery engine to perform this task.

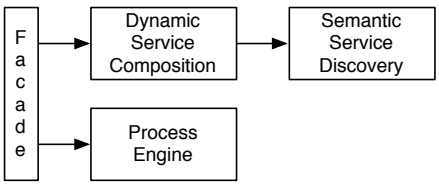
2.2 Solution Strategy: Automated Service Composition

Summary: Instead of modeling service compositions upfront and manually, they are created on-demand, based on the specific requirements of the current request. To achieve this, the goal of the request is expressed declaratively and the services are selected and bound together to adhere to the goal.

Applicability: As for late binding, automated service composition makes sense if service landscape changes are frequent or if adaptation to goal variations (see Section 3) is required. Service landscape changes can be arbitrary (no limitation to the exchange of just one service like for late binding) as long as all the required functionality is still available.

Consequences: The first consequence of using automated service composition is the necessity to specify a service request formally in a way that matches the specification of service functionality. In reality this can be tricky. Most of the time we cannot expect service users to specify formally what the composed service should achieve. In such a case other, e.g. template-based approaches are necessary. We also need formats to semantically specify request goals and service functionality. We already discussed briefly that the request goals need to be specified. This is required for *each* particular service invocation. Therefore the users need to specify their requirements, or we need a template into which users only have to fill in parameter values.

Additionally, using automated service composition has architectural consequences on the standard process-oriented SOA as displayed in Figure 1. Two components (Figure 3) are required to perform automated service composition: the component creating the composition and a semantic service discovery that assists the composer in creating the compositions. The responsibility of the discovery component is here not to find exactly matching services, but only to find services executable in a given state. Therefore discovery is only done on the preconditions, ignoring the effects. The according selection of services (considering the effects as well) is performed by the composer.



[htbp]

Fig. 3. Extended Process Engine for Automated Service Composition

Automated service composition can also affect enactment time negatively. Our experiences from ASG showed that the creation time for the composition

can be much longer than the actual enactment time. While some of these performance problems resulted from the prototypical implementation, composition creation time cannot be ignored. Especially in an environment where adherence to service level agreements is required, the usage of automated service composition needs to be investigated critically. Automated service composition can ensure adaptation to service landscape changes and to goal variations, and therefore reduces downtime and ensures better service provision to customers. But, the additional service delivery time can be contra-indicative.

Late binding and automated service composition can also be combined. This combination is quite promising. If automated service composition is used, most of the required functionality to perform late binding is already in place. The advantage of using both strategies in combination is that automated service composition only needs to be used if adaptation using late binding is not possible. Created service compositions are cached and if a suitable composition for a request is found, it is used and bound to concrete services. Minor changes to the service landscape can be overcome using late binding. Only if this approach is not successful, the cache is invalidated and new compositions are created. Automated service composition is still used to adapt to varying request goals.

As for late binding, adaptation for running service composition instances is not provided. If this is necessary, service re-composition (Section 4.2) can be used.

Implementation Sketch: To specify services and service request we used WSML [3] in the ASG project. As we can not expect end-users to model service requests in WSML, we used a template-based approach. A request template defines the basic requirements and the possible configurations. The template is then filled using values specified by the end-users in their actual service request.

Our approach for automatically created service compositions is based on existing work on automated planning. We are using an extended version of the enforced hill-climbing approach [4, 5] that allows for the composition of parallel and alternative control flows and the creation of objects [6]. Enforced hill-climbing is a search algorithm through state space that uses a heuristic for goal distance estimations in order to find a composition. Other approaches for automated service composition use approaches like model checking [7] or hierarchical task network planning [8].

3 Goal variations

The distinction between service landscape changes and goal variations is often blurred. This is due to the fact that similar solution strategies can be applied to both. Goal variations mean that different consumer requests demand service compositions with similar functionality but minor differences. The two classical solutions are to provide just one composition and users have to adapt their needs, or to provide individual compositions for each customer. While the first solution does not satisfy customers, the latter one increases provisioning costs. All variations need to be modeled and maintained. If the service landscape changes,

potentially all variations are affected. They need to be reviewed and modified if required.

A better approach is to not model all variations explicitly. In ASG, we used automated service composition (see Section 2.2) to handle goal variations. All variants are modeled declaratively. Instead of specifying which services needs to be invoked in what order, just the goal is defined. Introducing a new variant now does not require modeling a new process but just the service request. Managing variants has also become easier: if the service landscape changes, all variants adapt automatically. Often several variants can be merged into one request template. Each variant is then represented by another assignment of concrete values to the template.

Goal variations in ASG did not require an additional solution strategy. But other solution strategies are possible and can make sense if adaptation to service landscape changes is not required. In such a scenario, new developments to apply the product line approach to the management of process variations [9] can be used.

4 Service Failures

If not only own services are used, but also services by suppliers are integrated, dealing with service failures becomes important. As the external services are outside of our control, we can not guarantee that they are always available. But what should we do if a service fails? We could propagate the failure to the customer and compensate already invoked services. But this can be quite costly as well as unsatisfying for the customer. Instead we should try to adapt to the failed service by replacing it with another service or changing the structure of the composition. These two possible solution strategies are re-binding and re-composition.

4.1 Solution Strategy: Re-binding

Summary: If the invocation of a service fails, we halt the execution of the composition and replace the failed service with another service with equivalent functionality.

Applicability: Implementing re-binding makes sense if failures are frequent and we need to deal with them automatically without aborting the complete composition. As re-binding can be seen as delayed *late binding* operation, this functionality must be available. Hence, the same applicability constraints as for late binding (see Section 2.1) hold also here: it must be possible to determine service equivalence and the necessary adaptation must be limited to just the failed service. The approach additionally demands some failure detection mechanisms, which typically restricts the supported fault model to crash faults [10].

Consequences: As a consequence of using re-binding, we must be able to stop the execution of running compositions, change them, and restart them from the point we stopped it. Stopping a running composition has several implications: Does the enactment engine actually support stopping and restarting of compositions? How can we change a composition and feed the changes back into the engine? What do we do with running activities? Do we abort them or wait for their termination?

Another consequence is that we need to ensure that the failed service is not selected again. Re-selection of the same service is highly-probable as it previously was the optimal choice. But here it is counter-productive. The invocation will fail again, and we will perform re-binding until the service becomes available again (or forever).

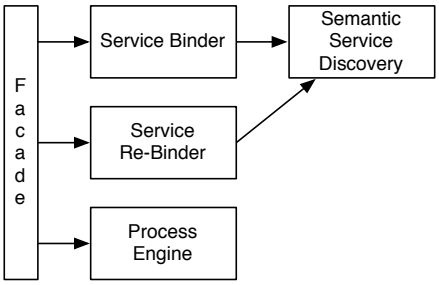


Fig. 4. Extended Process Engine for Re-binding

As Figure 4 shows, re-binding also requires an additional component. This component has the duty to find a suitable replacement if the invocation failed. As it has quite similar functionality like the service binder, both components are often merged. The interaction in case of a failure is like this: The process engine detects the failure and stops the enactment of the composition. It returns the current status of the composition to the facade, which in turn calls the re-binder to find a replacement. Afterwards the re-binder returns the updated composition to the facade. As the final step the facade hands the composition back to the process engine for re-enactment.

Implementation Sketch: How the different aspects of re-binding are implemented in ASG is quite similar to the implementation of late-binding. The only addition is a workflow engine that can stop the execution of a composition if the invocation of a service fails. Running service invocations are kept running and the changes are merged back into the running composition.

4.2 Solution Strategy: Re-composition

Summary: If the invocation of a service fails and re-binding is not possible, n the current state of the composition is used to create a new composition which is still able to reach the goal.

Applicability: Re-binding and re-composition make sense if frequent service invocation failures should be automatically handled. They are only applicable if automated service composition is available and if it is possible to determine the current state of an enacted composition. The latter means that given the knowledge about already executed services, their effects and the initial states are combined to form the initial state for the new composition.

Consequences: As automated service composition is used to perform re-composition, the same consequences apply here. Additionally, we must be able to stop the execution of running compositions, change them, and restart them from the point we stopped it. We also must prevent the usage of the failed service(s) in the new composition as it would otherwise fail again.

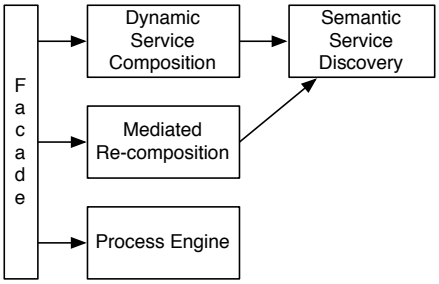


Fig. 5. Extended Process Engine for Re-composition

Figure 5 shows the architectural consequences of using re-composition. The main difference is an additional component: mediated re-composition. Mediated re-composition receives the stopped service composition and calculates the new initial state using the semantic service discovery and matchmaker. Afterwards the new request is given to the composer to create the updated service composition, which is finally enacted by the process engine.

Implementation Sketch: The implementation of re-composition in ASG mainly follows the implementation for automated service composition described earlier. Running service invocations are kept running but for the sake of automated service composition assumed to be finished. This ensures that no running invocations are removed from the updated compositions. The details of the approach are described in [11].

5 Dynamic QoS Requirements

Quality of Service (QoS) properties capture properties of services beyond their functionality. This includes for example service execution costs or execution duration demands. Service level agreements (SLA) are contracts between a service consumer and a service provider about concrete values of service quality properties. Consumer can have differing QoS requirements that are specified as part of their service request. The provider has to ensure that the provided service adheres to these requirements. If the provided service is a service composition, its QoS properties need to be deduced from the QoS properties of contained services.

If the QoS properties of services are known, QoS requirements by service consumers can be taken into account by late binding (Section 2.1) and/or automated service composition (Section 2.2) strategies. Services are selected and composed according to the QoS requirements. Another approach is to not store QoS properties of services as part of the service specifications, but to ask service providers dynamically. Going one step further, this allows us to negotiate with service providers about concrete values of several properties based on our current requirements and the providers offers.

5.1 Solution Strategy: Negotiation

Summary: Negotiating with service providers about concrete values of QoS properties allows for service level agreements (SLA) which are in line with the current requirements of the consumer and the provider's abilities. To perform negotiation, protocols like the iterated contract net interaction protocol [12] are usable.

Applicability: Obviously, the negotiation of QoS properties only makes sense if such requirements with regard to the service composition exist. They can either be explicitly specified by consumers in their request, or can be defined by the provider of the service composition. In the first case, negotiation allows for the adaptation to differing QoS requirements. In the second case, the adaptation to differing QoS properties of service providers is possible. Therefore, the second requirement for the negotiation strategy is the availability of QoS properties for services.

Consequences: Both the consumer of a service and the service provider need to provide negotiation functionality for this strategy. For the service provider, this means that each service needs to provide an additional interface just for negotiation. In addition, the service consumer needs to support similar functionality on his side to perform the negotiation. Components to perform the negotiation and to store and supervise the SLAs need to be established (see Figure 6). Negotiation is typically performed by an extended service binder. The SLA manager component stores all SLAs and monitors (in cooperation with the enactment engine) whether they are fulfilled or not. As a part of their interface, the protocol and data formats for negotiation and the format for SLA needs to be defined.

This must be seen as a non-trivial task, since the contract content data formats must be standardized between all participating parties.

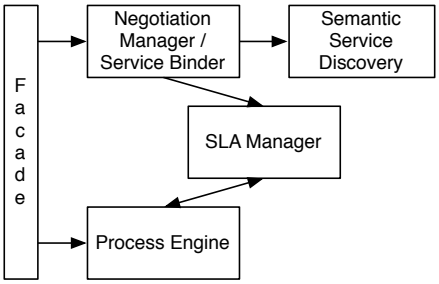


Fig. 6. Extended Process Engine for Negotiation

While late-binding can be used without negotiation, it is often used in combination with late binding (see Section 2.1), to ensure that we get the best SLAs from our partners. Which such a setup, we can negotiate with different providers of the same functionality and select among them the one with the best QoS properties.

One important aspect of negotiation is the machine-to-machine communication between different organizational domains. Each partner is interested in getting the best possible results out of these negotiations. Hence, partners might exploit weaknesses in the negotiation strategy of their counterpart. Detecting and avoiding such situations is very complicated and requires human involvement. This situation is quite similar to the search engine optimization (SEO) problem with which large search engine providers like Google, Yahoo!, and Microsoft are currently fighting. SEO is about modifying and extending Web pages in a way to ensure a higher ranking in these search engines resulting to attract more visitors and hence gain more revenue. To our knowledge, no general strategy to overcome these problems exists today. With negotiation, the problem is even more critical: If someone exploits a weakness in our negotiation strategy, we are not just 'visiting the wrong web page', but loosing money because of non-optimal contracts. We therefore argue that all negotiation requires some kind of frame contract among the negotiation parties, in order to ensure well-behaving and to give some bounds for negotiation values. In such an environment, the goal of negotiation is not to ensure the best possible results for oneself, but to find a satisfying mapping between the negotiation partners requirements and abilities.

As mentioned above, supervision of SLA fulfillment is crucial. Without it, the contracting of SLAs would be of limited use, as it is not known whether or not providers adhere to the contracted SLA. Hence, we need to constantly monitor concrete values and compare them to contracted values. In case of a violation, we need to react, which will be described in the next section.

Implementation Sketch: Negotiation in ASG was agent-based [13]. Both the service provider and the ASG platform implemented negotiation agents, who negotiated QoS properties implementing the iterated contract net interaction protocol.

6 Composition SLA violations

In ASG, two kinds of SLA occur: SLA for individual services and SLA for service compositions. SLA for individual services are contracted between the ASG platform as a service user and the provider of the individual service. Composition SLA are contracted between the end user and the ASG platform as the service provider. The ASG platform needs to monitor the fulfillment of individual service SLA in order to ensure the fulfillment of its own SLA with the end user. Hence, an important first step is the ability to actually monitor QoS properties.

The question is, what should happen if a service violates the contracted SLA? The simplest solution is to just propagate the violation to the end-user. But this can mean that we violate our SLA, too. Therefore other strategies are necessary if we want to customer confidence. For example, re-binding (Section 4.1) or re-composition (Section 4.2) can be extended to exchange services in case of a QoS violation.

7 Adaptation in ASG Scenarios

Scenarios in an EU-funded integrated project have to serve two different purposes. Industry partners are interested to find solutions for their concrete problems. On the other hand, scenarios should also demonstrate and motivate the research conducted in the project. Especially in large projects, there is not always a perfect match. With more than 10 industry partners and more than 10 research partners, this applied especially for ASG. Altogether, industry partners in the project provided 7 different scenarios from their daily work and industrial research interests. Two of them proved to be well suited as demonstration scenarios, since they show the applicability of nearly all research results and explain resulting benefits for the industry partners. We there now describe both scenarios in the context of the discussed solution strategies.

7.1 Attraction Booking

The idea behind the attraction booking scenario is to provide location-based services for mobile users in a telecommunication environment. The implemented functionality enables users to find attractions (e.g. events, theaters) based on their current position, to book tickets for them, and to find a route from their current location to the attraction. The service can be provided either by the mobile telephone provider or by a third party. Either way, it will depend on several external services:

– Localization services (internal for the mobile telephone provider)
– Attraction information services
– Attraction booking services
– Route planning services

Looking at the different scenarios of adaptation, we see that most of them fit to this scenario. The service provider will frequently integrate new attraction information and booking services. Hence, service landscape changes are typical. As external service are integrated, failures of services and violations of SLA are possible. Goal variations are not an issue here. All users consume the same service. Users of the attraction booking service also do not provide QoS requirements themselves, even though static QoS properties are considered for a minimal user experience.

7.2 Supply Chain Management for ISP

The second scenario was developed by an internet service provisioning (ISP) company. It provides domain registration and web space services to their customers, not only to end users, but also to resellers such as newspaper companies. The provided services are actually compositions of atomic internal and external services. The external services are:

– Domain checking services
– Domain registration services
– Payment services

To enter new markets, new services in all three categories need to be integrated in a quick and error-prone manner. Service landscape changes are quite frequent in this business environment. As the resulting service is not only provided to end users but also to resellers, goal variations occur with each new customer. Each reseller might have slightly different conditions (e.g. providing only an e-mail account) and preferences (e.g. a certain payment proceeding). The restrictions do not only apply to functional criteria, but also to QoS properties. This leads to dynamic QoS requirements for the overall service composition. As external service from third party providers (like Denic) are integrated, failures of services and violations of SLA are possible.

Table 1 gives an overview of the occurrence of the different adaptation scenarios on both ASG use cases. Most of the discussed cases can be observed in both scenarios, but the second scenario showed to be better suited to demonstrate ASG features, since all kinds of adaptation occur in it.

8 Implementation Remarks

The reference implementation of the Adaptive Service Grid platform was developed during the 30 months of project runtime, and is available as an open-source

Adaptation Scenario	Attraction Booking	Dynamic Supply Chain
Changing service landscape	yes	yes
Goal variations	no	yes
Service failures	yes	yes
Dynamic QoS requirements	no	yes
Composition SLA violations	yes	yes

Table 1. Adaptation scenarios and their impact on ASG scenarios

project[3] for public usage. Multiple demonstrations for both industry people and interested research partners showed that the practical implementation of our adaptation strategies works also in real scenarios.

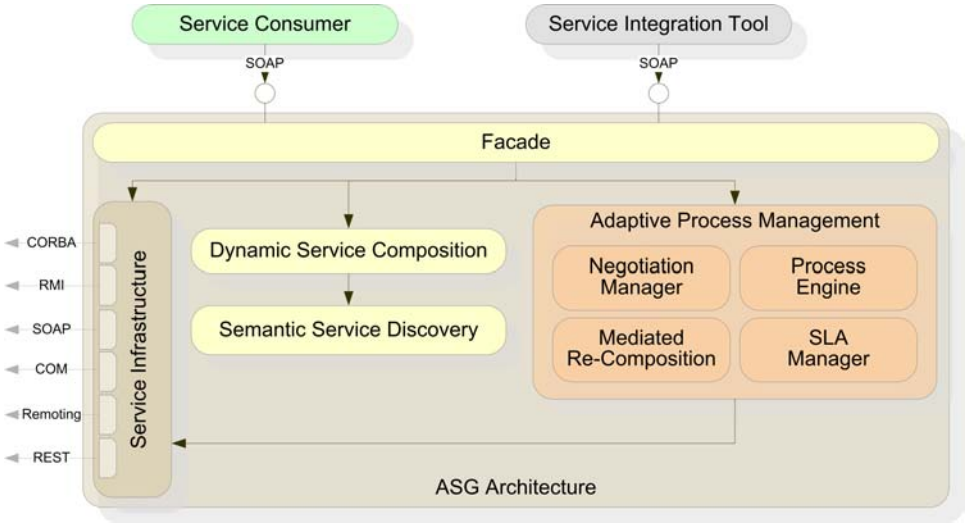


Fig. 7. ASG Reference Architecture

The platform implements all five solution strategies. It therefore contains components for dynamic service composition, semantic service discovery, service binding, negotiation, SLA management, re-binding and mediated re-composition. Figure 7 shows the architecture of the system. Service binding, re-binding and negotiation are all unified into the Negotiation Manager component. The components dealing with the enactment of compositions are all grouped into the *Adaptive Process Management* component.

The *Facade* component receives semantic requests for an application goal. This request is typically formulated by a front end application, which relies on ASG for the provision of a complex business functionality. The semantic request

[3] http://asg-platform.org/cgi-bin/twiki/view/Public/PrototypeDemo

is forwarded to the *Dynamic Service Composition* component, which collaborates with the *Semantic Service Discovery* to detect and combine feasible service candidates for a composition, based on semantic service descriptions. This is an application of the automated service composition solution strategy. Equivalent atomic services map to a similar semantic service description, making both of them an acceptable candidate for the later binding step. The abstract service composition is formulated in WS-BPEL with some enhancements to allow for the specification of semantically described services. The finalized composition is handed to the *Adaptive Process Management* (APM), which binds matching specific service from the current service landscape. The enactment of the resulting service composition is based on a *Service Infrastructure* which unifies the access and monitoring of services available in the service landscape.

In case of a service failure during enactment, ASG triggers a multi-stage recovery process based on the presented solution strategies. If the failure cannot be handled by the *Service Infrastructure*, the APM tries to perform a re-binding of another service fulfilling the same semantic sub-goal. In case of non-fatal errors like a SLA violation, the APM can also try to re-negotiate existing contracts about non-functional aspects. If this is also not possible, a re-composition is triggered in the higher level of the *Dynamic Service Composition* component. Here, ASG tries to find 'another way' using different services for reaching the aimed application.

9 Conclusion

In this paper we summarized adaptation strategies and scenarios considered and implemented in the ASG project. We identified five basic adaptation scenarios, namely service landscape changes, goal variations, service failures, dynamic QoS requirements, and composition SLA violations. We also introduced five solution strategies to deal with these scenarios. Late binding handles changes in the service landscape. Automated service composition handles changes in the service landscape as well as goal variations. Re-binding and re-composition both deal with service failures and composition SLA violations. And finally, negotiation handles dynamic QoS requirements. Figure 8 displays these relations graphically.

The adaptation scenarios and the according solution strategies were motivated in the project by two real world scenarios of the participating industry partners. The attraction booking scenario allows end users to find and book attractions. Especially the open service landscape led to the applicability of many of solution strategies. The dynamic supply chain for internet service providers scenario reflects the business of a company allowing its customers to register domain names and order web space. Here the open service landscape and the requirement to provide the service not only to end users but also to resellers lead to the applicability of all solution strategies.

The ASG project proposed a reference architecture that shows how all solution strategies can be combined into a coherent view. An implementation of this reference architecture was then used to demonstrate the actual applicability of

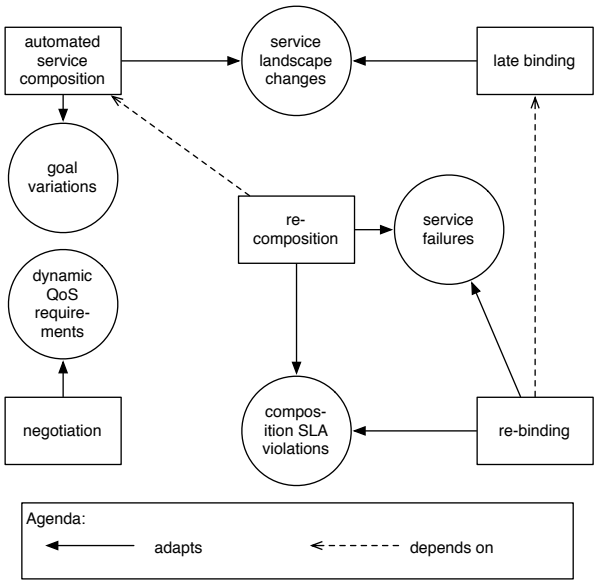


Fig. 8. Relations between adaptation scenarios and solution strategies

the solution strategies in the scenarios. This implementation is now released as open source and can be downloaded from the ASG project page.

Due to time constraints, ASG could not provide a quantitative analysis of the solution strategies. Questions about costs and overhead of the solution strategies (e.g. how high are upfront investments?) and about concrete benefits (e.g. are upfront investments at least balanced out by cost savings?) had to remain unanswered, and will be part of future research efforts.

References

1. Haller, A., Cimpian, E., Mocan, A., Oren, E., Bussler, C.: Wsmx—a semantic service-oriented architecture. In: Proceedings of the International Conference on Web Service (ICWS 2005), Orlando, Florida (2005)
2. Verma, K., Gomadam, K., Sheth, A.P., Miller, J.A., Wu, Z.: The meteor-s approach for configuring and executing dynamic web processes. Technical report, LSDIS LAB, University of Georgia, Athens, Georgia (2005)
3. de Bruijn, J., Lausen, H., Krummenacher, R., Polleres, A., Predoiu, L., Kifer, M., Fensel, D.: D16.1v0.21 web service modeling language (wsml) (2005)
4. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research **14** (2001) 253 – 302
5. Hoffmann, J., Brafman, R.: Contingent planning via heuristic forward search with implicit belief states. In: Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05), Morgan-Kaufmann (2005)
6. Meyer, H., Weske, M.: Automated service composition using heuristic search. In Dustdar, S., Fiadeiro, J.L., Sheth, A., eds.: Business Process Management (BPM

2006). Volume 4102 of Lecture Notes In Computer Science., Heidelberg, Springer (2006) 81–96
7. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and monitoring web service composition. In: Workshop on Planning and Scheduling for Web and Grid Services (held in conjunction with The 14th International Conference on Automated Planning and Scheduling. (2004) 70 – 71
8. Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D.: HTN planning for web service composition using shop2. Journal of Web Semantics **1** (2004) 377–396
9. Schnieders, A., Puhlmann, F.: Variability modeling and product derivation in e-business process families. In: Technologies for Information Systems, Springer (2007) 63–74
10. Jean-Claude Laprie and Brian Randell anf Carl Landwehr: Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions On Dependable And Secure Computing **1** (2004)
11. Gajewski, M., Meyer, H., Momotko, M., Schuschel, H., Weske, M.: Dynamic failure recovery of generated workflows. In: DEXA Workshops, IEEE Computer Society Press (2005) 982–986
12. Foundation for Intelligent Physical Agents (FIPA): FIPA Iterated Contract Net Interaction Protocol Specification. (2002)
13. Momotko, M., Gajewski, M., Ludwig, A., Kowalczyk, R., Kowalkiewicz, M., Zhang, J.Y.: Towards adaptive management of qos-aware service compositions. nternational Journal on Multiagent and Grid Systems **4294** (2007) 637–650