Chapter 1
Design and Management of Web Service Transactions with Forward Recovery

Peter Dolog, Michael Schäfer and Wolfgang Nejdl

Abstract In this chapter we describe a design of compensations using forward recovery within Web service transactions. We introduce an approach to model compensation capabilities and requirements using feature models, which are the basis for defining compensation rules. These rules can be executed in a Web service environment that we extend with the concept of an abstract service, which is a management component for flexible compensation capabilities. We describe the design and also discuss advantages and disadvantages of such an approach.

1.1 Introduction

A Web service allows a *provider* to encapsulate functionality and to make it available for use via a network. A *client* can invoke such a Web service to use its functionality. By combining existing Web services from different service providers, a new and more complex *distributed application* can be created, which in turn can be offered as a new value-added *composite service*. Such a distributed application is usually created based on a *business process*, which consists of a logical sequence of actions that can include the invocation of a Web service. Accordingly, it is vitally important to control the processing of each single action and the overall process in order to be able to guarantee correct execution. This is done by using *transactions*.

P. Dolog (✉)
Department of Computer Science, Aalborg University, Selma Lagerlöefs Vej 300, 9220 Aalborg, Denmark
e-mail: dolog@cs.aau.dk

M. Schäfer · W. Nejdl
L3S Research Center, University of Hannover, Appelstr. 9a, 30167 Hannover, Germany
e-mail: Michael.K.Schaefer@gmx.de

W. Nejdl
e-mail: nejdl@l3s.de

A. Bouguettaya et al. (eds.), *Advanced Web Services*,
DOI: 10.1007/978-1-4614-7535-4_1,

A transaction consists of a set of *operations* ("units of work") that are executed within a system. Before and after the transaction, this system has to be in a consistent state [6]. The concept of transaction originates from database systems, which require an effective control of operations in order to guarantee data consistency. This is achieved by requiring that transactions fulfill the ACID properties [6, 7]: *Atomicity*, *Consistency*, *Isolation*, and *Durability*.

In the context of a distributed application, a *distributed transaction* [6] controls the execution of operations on multiple loosely-coupled Web services (*participants*) from different providers. Each operation is an invocation of one of the services and executes functionality provided by the particular service that is called. Any kind of service, independent of the actual functionality it implements (e.g. reserving a flight, performing a money transfer, transforming data), can in principle participate in such a transaction. A *coordinator* is responsible for the creation of the transaction, the registration of participants, and the evaluation of the participant's results.

Due to the fact that a distributed transaction has to include external sources via a network connection, it is usually not possible to fulfill all ACID properties, as each one imposes restrictions on the system which can be a disadvantage in such an environment. For example, in order to be able to handle long-running transactions, which take a long time until they complete, it is necessary to relax the isolation property. This means that locks on resources are removed even though the overall transaction is not yet complete, so that other transaction can access these resources and are not blocked. However, it can still happen that the transaction fails, and if this is the case it is necessary for the coordinator to initiate a *compensation*, which reverts all operations that were already performed in order to restore the state of the system before the transaction was started. The Web services that were already processed have to do a *rollback*, i.e. they have to execute a predefined set of actions that undo their original operation. This notion of rolling back the system to a previous state is known as *backward recovery* [16], as it reverses the operations that have been performed. Whether such rollback operations exist, and what steps they consist of, depends highly on the system and the Web services involved. A compensation protocol can only provide the orchestration of compensative activities, the developer of a rollback operation has to ensure that its result represents the consistent state before the transaction was started.

There are alternative approaches how to relax the isolation property within a Web service environment. Reference [9] describes the "Promises" pattern, which defines a "Promise Manager" that receives resource promise requests from a service. In comparison to the classic ACID isolation, this promise does not lock an individual resource but instead ensures that one from a pool of (anonymous) resources with the same properties will be available.

Web service coordination and transaction specifications [11–13] have been defined that provide the architecture and protocols required for transaction coordination of Web services. Several extensions have been proposed to enhance these protocols to add more flexibility [2, 20]. While these protocols provide the means for transactions in a distributed environment, it is still a challenge to guarantee its consistency.

[10] describes an approach to check already at application design time whether the distributed application will always terminate in a consistent state.

The specifications in their original form provide only limited compensation capabilities [8]. In most cases, the handling of a service failure is restricted to backward recovery. Subsequently, the aborted transaction will usually have to be restarted, because the failed distributed application still has to perform its tasks. Backward recovery therefore results in the loss of time and money, and additional resources are needed to restart the transaction. Moreover, the provider of the service that has encountered an error might have to pay contractual penalties because of a violated *Service Level Agreement* (*SLA*). The rollback of the complete transaction due to the failure of one service can also have widespread consequences: All *dependent transactions* on the participating Web services (i.e. transactions that have started operations on a service after the currently aborting transaction and therefore have a completion dependency [3]) have to abort and perform a rollback, which in turn can trigger the abort of other transactions and thus lead to cascading compensations. This is sometimes called the *domino effect* [16].

In addition to the problematic consequences of backward recovery, current approaches do not allow any changes in a running transaction. If for example erroneous data was used in a part of a transaction, then the only possible course of action is to cancel the transaction and to restart it with correct data.

An alternative to backward recovery is *forward recovery*. The goal of this approach is to proactively change the state and structure of a transaction after a service failure occurred, and thus to avoid having to perform a rollback and to enable the transaction to finish successfully.

To illustrate forward recovery in a Web service environment, consider as example a company's monthly payroll processing. In the first step, the company has to calculate the salary for each employee, which can depend on a multitude of factors like overtime hours or bonuses. In the next step, the payment of the salary is performed, which comprises several operations: Transfer of the salary from the company's to the employee's account, transfer of the employee's income tax to the account of the fiscal authorities, and printing and mailing of the payslip. The employee has only one task, which he has to perform each month: He transfers the monthly installment for his new car to the car dealer's account.

The company's and the employee's operations are each controlled by a business process and are implemented using services from multiple providers (Fig. 1.1). The two business processes use transactions in order to guarantee a consistent execution of all required operations. For simplicity, only the services of transaction T1 are shown, although of course also transaction T0 and T2 consist of several services.

While this scenario is quite simple, it has multiple dependencies within and between the two running transactions. Therefore, it is important that both transactions can complete successfully and do not have to be aborted and rolled back. Nevertheless, a situation in which such a need arises can become imminent quite easily:

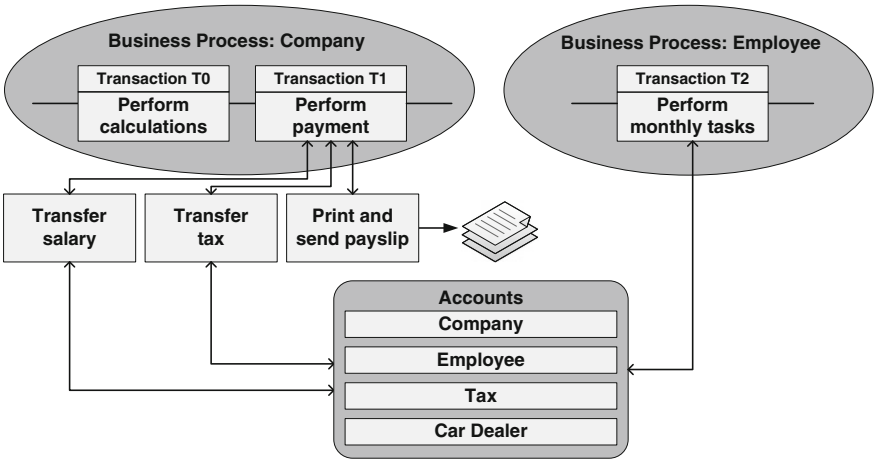

Fig. 1.1 The example scenario

- Situation 1: One of the Web services might encounter a problem during the execution of its operations. For example, it could be that the service that transfers the salary fails due to an internal error.
- Situation 2: A mistake might have been made regarding the input data of one of the operations. In this scenario, it could be that the calculation of the salary is flawed and too much has been transferred to the employee's account.

As explained above, using a backward recovery approach in such a scenario would be costly. However, using a forward recovery approach allows to handle both situations without a rollback:

- Situation 1: Although the Web service failed, it would still be possible to save the transaction by using a different service. Such a *replacement* of the original Web service is encouraged by the fact that usually multiple services from different providers exist that provide the same or similar capabilities.
- Situation 2: The operations with the erroneous input data have already been processed, and the transaction would have to be rolled back even if an administrator notices the failure before the transaction has been finished. However, the salary transfer could be easily corrected with another money transfer operation.

This scenario is only one example where a forward recovery of transactions would be beneficial. Similarly, such an approach could help in other situations such as overloaded services, timeouts, or other errors.

In this chapter we describe a design approach [17] and an environment which is able to handle forward recovery compensation of Web service transactions [19]. The approach is based on the idea that there is a possibility to replace a failed service in the transaction with another one with the same or similar capabilities and by doing so to avoid unnecessary rollbacks. In addition, the design includes an approach to model and match compensation capabilities and requirements. The contribution

of this chapter is that it provides more detailed examples and explains the whole approach from design of the rules to their execution within the environment.

The main idea of the design is the introduction of a new component called an *abstract service*, which does not directly implement any functionality that is provided to the client, but instead functions as a management unit for flexible compensation capabilities [18]. As part of these capabilities, it specifies and manages potential replacements for participating Web services to be used. The compensations are performed according to predefined *rules*, and are subject to *contracts* [14]. An abstract service's functional and compensation capabilities can be specified using *feature models*, which allow a client to describe his requirements for a service, and a provider to specify a service's capabilities. These individual feature models can be used to automatically find matching services for a given set of requirements.

Such a solution has the following advantages:

- Compensation strategies can be defined on both, the service provider and the client side. They utilize local knowledge (e.g. the provider of a service knows best if and how his service can be replaced in case of failure) and preferences, which increases flexibility and efficiency.
- The environment can handle internally and externally triggered compensations.
- The client of a service is informed about complex compensation operations, which makes it possible to trigger additional compensations. Compensations can thus consist of multiple operations on different levels.
- By extending the already adopted Web service specification, it is not necessary to discontinue current practices if compensations are not required.
- The separation of the compensation logic from the coordination logic allows for a generic definition of compensation strategies, independent from the coordination specification currently in use. They are therefore more flexible and can easily be reused in a different context.

The rest of the chapter is structured as follows: Sect. 1.2 describes how we propose to represent compensation capabilities using feature models. Section 1.3 describes the specification of compensation rules and possible compensation activities. Section 1.4 describes the abstract service architecture where compensations can be executed according to compensation rules. Finally, Sect. 1.5 provides a discussion regarding advantages and shortcomings of the approach.

1.2 Compensations Design

We are introducing a compensation design approach which provides a set of models that describe both functional and compensation capabilities of a service:

- on the service provider side, *mandatory features* which are needed to provide at least the minimum functionality, as well *optional features* which extend the capabilities or level of service;
- on the service consumer side, features the client requires in order for the service to suit his needs.

We adopt a feature modeling approach and a methodology from [5, 17]. According to that methodology, first a *conceptual model* is defined which describes the main concepts and relationships between them. The configuration view on the concepts is described by means of *feature modeling* for both functionality and compensation capabilities.

Subsequently, the functionality and compensation models are merged to describe the offered capabilities by a service provider, or requested functionalities and restrictions posed on compensations by a service consumer. Different algorithms can then be employed to match feature models of a client and a service provider. In the following we will explain the introduced models in more detail.

1.2.1 Conceptual Model

In order to formalize different types of compensations, a conceptual model has been created that constitutes the basis for the feature models in the following sections. The result is the *compensation concept model* as seen in Fig. 1.2.

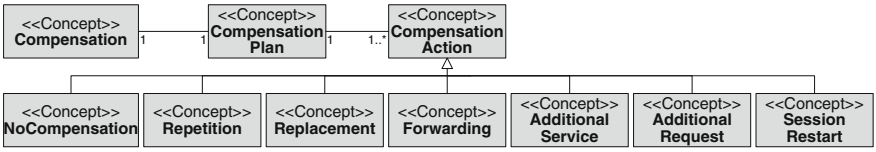

Fig. 1.2 The compensation concept model

Each *Compensation* contains a *CompensationPlan*, which in turn consists of one or more *CompensationActions*. Which CompensationActions exist and how they can be implemented depends on the actual environment. Accordingly, the ones listed are not necessarily complete and can be extended in the future.

Based on this definition of compensation concepts, it is now possible to create feature models in order to define what a service *can do*, *should be able to do*, and *is not allowed to do*.

1.2.2 Compensation Feature Model

The compensation concept model is the basis for the definition of the *compensation feature model*, which is depicted in Fig. 1.3. It describes the mandatory and optional features of the compensation concept, and will be used in the next step to define service-specific feature models, which can be part of a contract between a service provider and a service client.

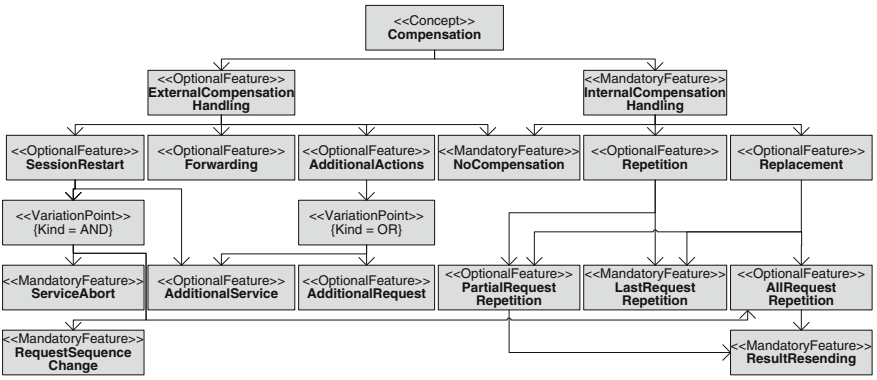

Fig. 1.3 The compensation feature model

The main two features of the model are the *InternalCompensationHandling* and the *ExternalCompensationHandling* features. An internal compensation is triggered by an internal service error, while an external compensation is triggered from outside of the transaction. An example for an externally triggered compensation could be the handling of the salary transfer mistake from the scenario described in Sect. 1.1 that is spotted by an administrator.

These main features structure the available compensation types as features according to their application: *Repetition* and *Replacement* are only available for internal compensation operations, and *SessionRestart*, *Forwarding* and *AdditionalActions* are only available for external compensation operations. The exception for this separation is *NoCompensation*, which is the only common compensation feature. Only two of these features are mandatory, the *InternalCompensationHandling* and the *NoCompensation* feature. This is due to the fact that the elementary feature of a compensation in our context is inactivity: If no rule or compensation capabilities exist, then the service has to fail without any other operations. Accordingly, the ability to perform external compensations is only optional.

The *SessionRestart* includes as an optional feature the invocation of an additional service (*AdditionalService*), and requires via a variation point (AND) the *ServiceAbort*, *RequestSequenceChange*, and *AllRequestRepetition* features. The capability to abort the service, change the request log, and resend all requests is needed to perform the session restart, and therefore these three features are mandatory. Likewise, the *AllRequestRepetition* feature cannot work without the *ResultResending* feature.

Within an externally triggered compensation, it is possible to invoke additional services and to create and send additional requests to the service. That is why the *AdditionalActions* feature includes the *AdditionalService* and *AdditionalRequest* features. They are connected via an OR variation point as the *AdditionalActions* feature needs at least one of these two features.

The basic operation mode of the *Repetition* compensation feature is the resending of the last request to the service. Therefore, the *LastRequestRepetition* feature is

mandatory, and the *PartialRequestRepetition* only optional. Finally, the *Replacement* feature requires at least one of the *LastRequestRepetition*, *PartialRequestRepetition*, or *AllRequestRepetition* features.

1.2.3 Capability Feature Model

The *Capability feature model* specifies the capabilities of a service. This model can be provided in the public description of the service (e.g. in the UDDI entry), and can thus be used in the client's search process for a suitable service.

The definition of a service's features includes both the specification of functionality as well as compensation capabilities. The capability feature model is realized by merging the service's functionality feature model with its compensation feature model. The *functionality feature model* describes the features of the service that constitute the offered operations, e.g. the booking of a flight. The compensation feature model describes the service's compensation capabilities. It is created by using the compensation feature model described in the previous section as a basis, and then altering it by deleting features that are not offered (e.g. a service that does not provide external compensation capabilities will delete this part of the model completely), by changing the mandatory/optional properties, or by adding features at certain parts (e.g. by specifying the additional services that can be used in the compensation process).

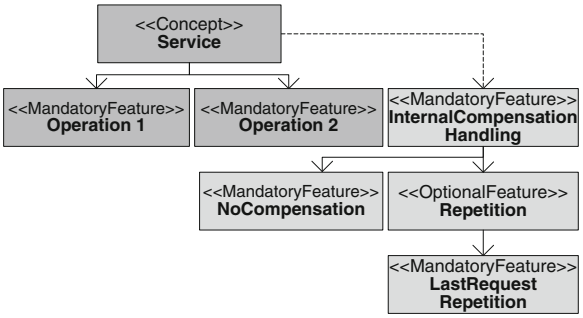

Fig. 1.4 Merging of the functionality and compensation models

This process of merging the two different models is depicted in Fig. 1.4. Here, a service offers two basic operations, "Operation 1" and "Operation 2", which form the functionality feature model (dark grey). The service is able to handle internal compensations by either doing nothing (the mandatory default action), or by repeating the last request. This forms the compensation model (light grey). The two models are merged (symbolized by the dashed arrow), and thus form the service's capability feature model. The mandatory/optional properties are interpreted in this context as "will be used by the service" and "can be used by the service", respectively. The interpretation is different in the requirement feature model.

1.2.4 Requirement Feature Model

The client creates a requirement description in order to be able to initiate a search for a suitable service. The specification is being done very much like the definition of the capability feature model described in the previous section: A common model is being created that includes the required functionality and compensation features. This model is called the *requirement feature model*.

However, although the basic process of creating the requirement feature model is the same, the interpretation of the mandatory/optional properties differs. A mandatory feature *has* to be provided by the service and is thus critical for the comparison process, while an optional feature *can* be provided by the service, and is seen as a bonus in the evaluation of the available services.

In the search process, each service's capability feature model will be compared to the client's requirement feature model, and the client can thus decide which service is suitable for its needs.

1.2.5 Restriction Feature Model

After the client has found the necessary services that offer the required functional and compensation features, the contract negotiation with each service will be performed. A vital part of this contract is the specification of compensation features that the service is allowed to use. While it is of course possible to do this restriction by simply searching for services that only perform the allowed compensation actions, such an approach significantly reduces the available services. Moreover, it is quite possible that a client wants to use the same service in multiple applications, where each has its own rules regarding the compensation actions that are permitted. Therefore, it is beneficial to use a *restriction feature model* that can be part of the contract, and to which the service dynamically adapts its compensation actions.

The restriction feature model can be defined by using the compensation feature model described in Sect. 1.2.2. By removing features from this model, the client can state that these are not allowed to be used in the compensation process. Only those features that are still in the model are permitted. Therefore, it is not necessary to distinct between mandatory and optional features.

When the service wants to invoke a specific compensation action, it will first consult the contract's restriction feature model. If the compensation action is part of the model, then the service is allowed to use it. This way, the service can dynamically adapt to the requirements of each single client.

1.2.6 Model Comparison Algorithm

We define a comparison algorithm to match the requirement model of a client and the capability model of a service. These two models are the input for the

algorithm, which iteratively compares them and calculates a numerical *compatibility score*:

- Using the requirement feature model as a basis, the features are compared stepwise. In this process, it is required that the same features are found in the same places, because the same feature structure is expected.
- Each mandatory feature in the requirement model has to be found in the capability feature model. If this is not the case, the comparison has failed and a negative compatibility score is returned to indicate this. However, if a mandatory feature is included in the capability model, this will not have any impact on the comparison score, as the mandatory features are the minimum this is expected.
- Each optional feature in the requirement model can exist in the capability model, but does not have to. Each one found counts as a bonus added to the compatibility score. This accounts for the fact that a service that provides more than the minimum is better, as it can more easily be used in different applications.
- Additional features in the service's capability model, like the specification of additional services used in the compensation process, are ignored as long as they are found in the appropriate place, e.g. as a subfeature to the "AdditionalService" feature. Any other additional features will lead to a failure of the comparison.

Once the comparison is completed, the compatibility score will be returned. At the moment, a very simple score is used that does not include advanced properties like feature priorities, which could be used in the future:

- If the comparison has failed, the compatibility score will be -1.
- Each mandatory feature that is found does not increase the score. A service which provides only the mandatory features (and is thus suitable) will therefore have a compatibility score of *0*.
- Each optional feature in the capability model increases the score by *1*.

As it can be seen, every compatibility score equal to or higher than 0 classifies a service as suitable for the client's needs. Moreover, the higher the score is, the more features a service provides. Using this simple score, it is easy to compare multiple services and their capabilities.

1.2.7 Example

The use of feature models will now be examined based on the "Transfer salary" service of the scenario described in Sect. 1.1. The services in this scenario can be used in different distributed applications, and it is therefore important that their compensation capabilities can be adapted.

Capability Feature Model (depicted in Fig. 1.5): The functional features of this service are the "Debit" and "Credit" operations, which are mandatory. The service is capable of performing all compensation actions, and accordingly the complete compensation feature model is merged with the functional model. Finally, the service

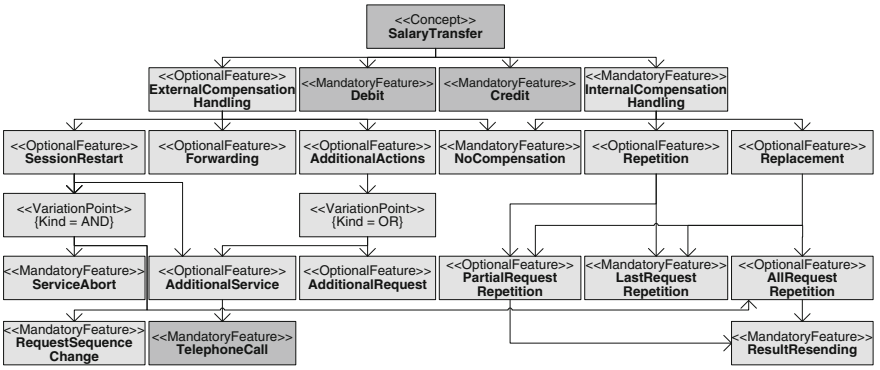

Fig. 1.5 The SalaryTransfer capability feature model

specifies that an additional service will be used in the compensation procedures: This is defined by adding the "TelephoneCall" feature to the "AdditionalService" feature. By providing this feature model, the service can state its capabilities and informs the client about a special operation it uses for this purpose.

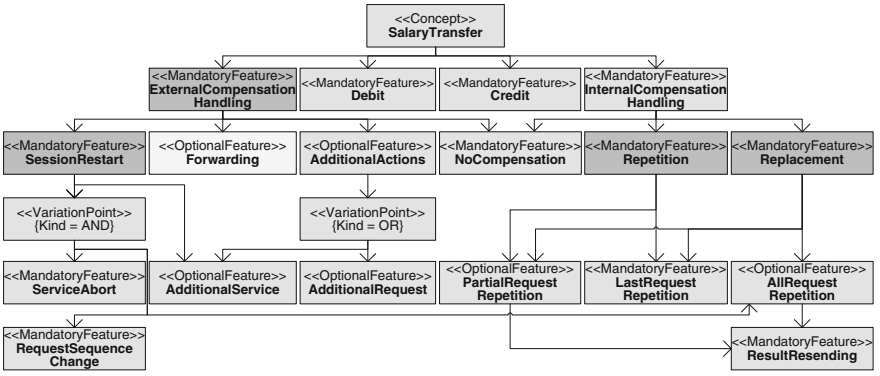

Fig. 1.6 The SalaryTransfer requirement feature model

Requirement Feature Model (Fig. 1.6): The client that creates the payment processing application specifies its requirements for the "Salary Transfer" service in a requirement feature model. The functional features are the "Debit" and "Credit" operations. Regarding the required compensation features, the client is looking for a service that is able to perform the "Repetition" and "Replacement" compensation actions for internal error handling, and the "SessionRestart" for external compensation handling. Accordingly, these features are marked as "mandatory".

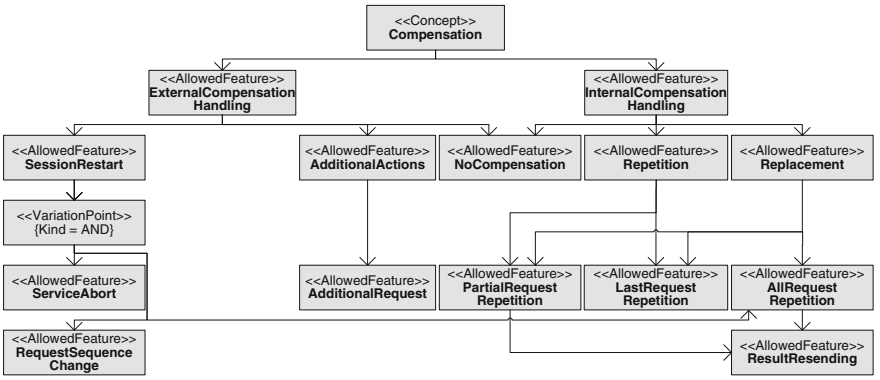

Fig. 1.7 The restriction feature model

Restriction Feature Model (Fig. 1.7): After the client has found a suitable service that offers the required capabilities, he defines the permitted compensation actions. In this example, the client does not want for the new application's service that it uses additional services in the event of a compensation. Therefore, the respective feature ("AdditionalService") is removed from the compensation feature model. This restriction model is part of the contract that the client has with the service. When the service now encounters a situation that requires compensation, it will only execute compensation plans that are in accordance with the model's restrictions.

1.3 Compensation Rules

Compensation rules are specifications of permitted compensations in the context of a particular Web service. The compensation activities and types that are part of these rules are adopted by a designer from the compensation and capabilities feature models. Two different kinds of compensations can be specified within these rules: Internally triggered compensations, which can be handled through a service replacement, and externally triggered compensations.

Each rule specifies the exact steps that have to be performed in the compensation process. For the purpose of defining the available compensation operations, we distinguish between basic *compensation activities*, which constitute the available single compensation operations, and complex *compensation types*, which are composed compensation processes consisting of multiple activities. This is shown in Fig. 1.8.

The compensation types specify which *combinations* of compensation activities can be defined in rules for handling internal and external compensations, as it is not desirable to allow every possible combination within the environment. When a service receives a request for a compensation, it will first of all check whether a rule for the current situation exists, and if this is the case, it will validate each rule before executing the given set of compensation activities in order to guarantee that they are consistent with the available compensation types.

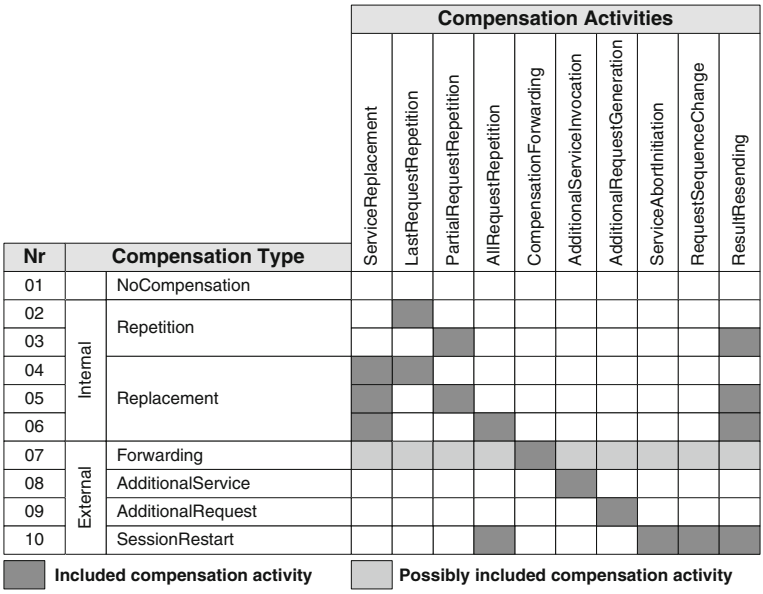

Fig. 1.8 The compensation types and their included activities

Therefore, although the combination of different compensation activities allows the definition of flexible and complex rules, it is not permitted to define arbitrary compensation handling processes. Only the predefined compensation types can be used, and it is thus guaranteed that a service does not execute a process defined in a compensation rule that is not permitted or possible. At the same time, this approach allows the future extension of the environment with new compensation strategies: In order to test or include new compensation strategies, it is possible to simply define a new compensation type and extend the service to accept it.

1.3.1 Basic Compensation Activities

Compensation activities are the basic operations which can be used in a compensation process. *ServiceReplacement* replaces the currently used Web service with a different one, which offers the same capabilities. *LastRequestRepetition* resends the last request to the service. *PartialRequestRepetition* resends the last *n* requests from the request sequence of the current session (i.e. within the current transaction) to the service, while *AllRequestRepetition* resends all requests. *CompensationForwarding* forwards the external compensation request to a different component that will handle it. *AdditionalServiceInvocation* invokes an additional (external or internal) service, which performs a particular operation required for the compensation

(e.g. the invocation of a logging service). *AdditionalRequestGeneration* creates and sends an additional request to the Web service. Such a request is not influenced by the client, and the result will not be forwarded to the client. *ServiceAbortInitiation* cancels the operations on the service, i.e. the service aborts and reverses all operations which have been performed so far. *RequestSequenceChange* performs changes in the sequence of requests that have already been sent to the Web service. *ResultResending* sends new results for old requests, which have already returned results.

1.3.2 Compensation Types

Compensation types aggregate multiple compensation activities, and thus form complex compensation operations (Fig. 1.8). These types are the compensation actions which can be used for internal and external compensations.

The most simple type is *NoCompensation*, which does not perform any operation.

The *Repetition* type is important for the internal error handling, as it repeats the last request or the last *n* requests. The last request can for example be resent to a service after a response was not received within a timeout period. A partial resend of *n* requests can for instance be necessary if the request which failed was part of a sequence. A partial repetition of requests will result in the resending of results for old requests to the client, which has to be able to process them.

The compensation type *Replacement* can be used if a service fails completely. It replaces the current service with a different one, and resends either all requests, a part of the requests, or only the last one. Resending only the last request is possible if a different instance of the service that has failed can be used as replacement, which works on the same local data and can therefore simply continue with the operations.

Forwarding is special in comparison with the other types as it only indirectly uses the available activities. It forwards the handling of the compensation to a different component, which can potentially use each one of the compensation activities (which are therefore marked as "possibly included") in the process.

In an externally triggered compensation, it is sometimes necessary to invoke additional services and send additional requests to the service. For this purpose, the compensation types *AdditionalService* and *AdditionalRequest* exist.

The final compensation type is *SessionRestart*. This operation is required if the external compensation request cannot be handled without a restart of the complete session, i.e. the service has to be aborted and subsequently the complete request sequence has to be resent. The requested change will be realised by a change in the request sequence prior to the resending.

1.3.3 Example of a Compensation Rule

Figure 1.9 shows an example of an external compensation rule specified in an XML document. This example rule handles the refund of excess salary that has been transferred to the employees account as described in the example in Sect. 1.1.

```
<cmp:ExternalCompensationRule identifier="refundSalaryDifference">
  <cmp:CompensationCondition>
    <cmp:RequestMethod identifier="transferSalaryMethod" />
    <cmp:ParticipantRequest identifier="getAccountBalanceMethod"
      parameterFactory="CheckEmployeeAccountParameterFactory">
      <cmp:Result resultEvaluator="AccountInCreditResultEvaluator" />
    </cmp:ParticipantRequest>
  </cmp:CompensationCondition>
  <cmp:CompensationPlan>
    <cmp:Compensation>
      <cmp:AdditionalRequest identifier="transferSalaryMethod"
        parameterFactory="RefundSalaryDifferenceParameterFactory" />
    </cmp:Compensation>
    <cmp:Compensation>
      <cmp:ServiceRequest
        serviceAddress="http://localhost:8080/axis/services/TelephoneCall"
        methodName="initializeTelephoneCall" />
    </cmp:Compensation>
  </cmp:CompensationPlan>
</cmp:ExternalCompensationRule>
```

Fig. 1.9 An example compensation rule

The compensation condition consists of two single condition elements:

1. *RequestMethod*—The rule applies to external compensation requests, which aim at changing requests that originally invoked the service's method "transferSalaryMethod", i.e. it applies to external compensations that try to change the details of an already completed salary transfer.
2. *ParticipantRequest*—The second condition element specifies a request that has to be sent to the current service. The goal of the request is to check whether the account of the employee will still be in credit after the excess amount has been refunded. The condition's request invokes the service's method "getAccountBalanceMethod". The request parameters are created by the parameter factory "CheckEmployeeAccountParameterFactory". After the request has returned the current balance, the predefined result evaluator "AccountInCreditResultEvaluator" is responsible for checking whether the salary refund can be performed, and thus whether the rule's condition is fulfilled or not.

The rule's compensation plan consists of two steps as well:

1. *AdditionalRequest*—An additional request is sent to perform the required changes, i.e. the money transfer back to the company's account. It invokes the service's method "transferSalaryMethod". The parameters for this request are created by the parameter factory "RefundSalaryDifferenceParameterFactory".
2. *ServiceRequest*—An additional external service is used as part of the compensation. The method "initializeTelephoneCall" has to be invoked. This external service performs a precautionary telephone call which informs the employee about the error in the salary calculation and the refund that has been performed.

This is of course only a simple example. External compensation rules can consist of a multitude of single conditions and/or compensation operations.

1.4 Web Service Environment with Transaction Coordination

The compensation rules from the previous section can be interpreted by an environment we have designed and implemented as a prototype. The environment builds upon adapted Web service coordination and transaction specifications [11–13]. They provide a conceptual model and architecture for environments where business activities performed by Web services are embedded in a transactional context.

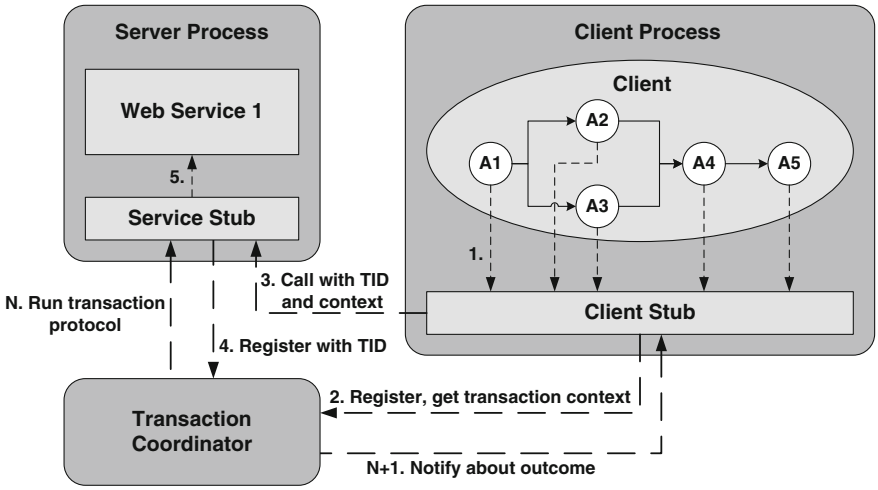

Fig. 1.10 Transactional environment for Web services adopted from [1]

Figure 1.10 depicts an excerpt of such an environment with the main components. The client runs business activities A1 to A5, which are part of a transactional context that is maintained by a transaction coordinator. Client and server stubs are responsible for getting and registering the activities and calls for Web services in the right context. The sequence of conversation messages is numbered. For clarity, we only show a conversation with a Web service provider that performs business activity A1. The coordinator is then responsible for running appropriate protocols, for example a distributed protocol for Web service environments such as [2].

We extend the architecture and the infrastructure based on the specifications [11–13] in order to enable it to handle both internally and externally triggered compensations as described in the previous sections.

Figure 1.11 depicts the extension to the transaction Web service environment, namely the *abstract service* and the *adapter* components. This extension does not change the way how client, coordinators and providers operate. Instead of invoking a normal Web service, a client invokes an abstract service, which looks like a standard Web service to the outside. However, the abstract service is a management component for forward recovery compensation handling, which wraps multiple *con-*

crete services that offer the same functionalities and can thus replace each other. The abstract service is therefore a *mediator* between a client and the concrete service that performs the required operations. At the same time, the adapter functions as a mediator between transaction coordinator, abstract service and concrete service to ensure proper transactional context and to provide the means to intercept failure notifications and create messages required in the compensation handling process.

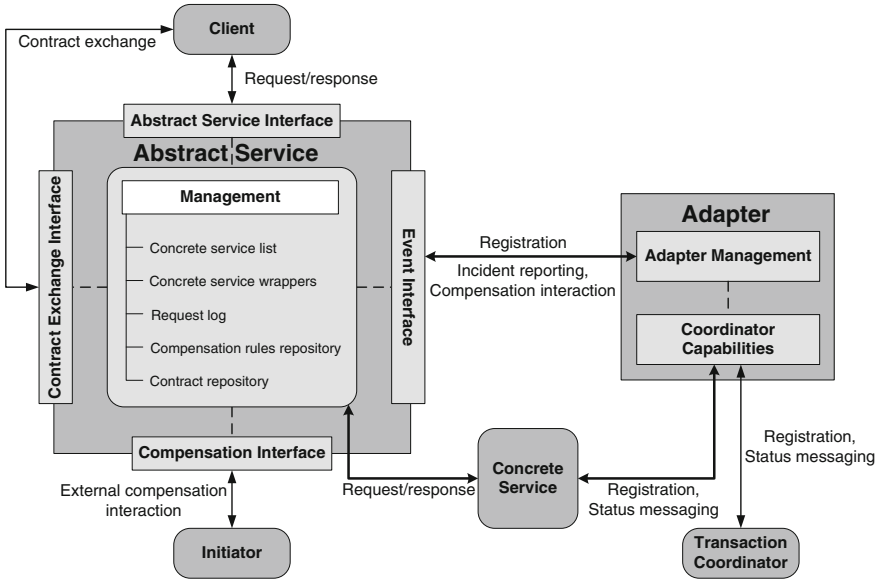

Fig. 1.11 The abstract service and adapter transaction environment

1.4.1 Abstract Service

The central element of the extension is the notion of the *abstract service*. The client stub communicates with the Web service provider stub through the abstract service. An abstract service does not directly implement any operations, but rather functions as a management unit, which allows to:

- define a list of Web services which implement the required capabilities,
- invoke a service from the list in order to process requests which are sent to the abstract service,
- replace a failed service with another one from the list without a failure of the transaction, and
- process externally triggered compensations on the running transaction.

To the outside, it provides an abstract interface and can be used like any other Web service, and uses the same mechanisms like SOAP [15] and WSDL [4]. On the inside,

it manages a list of concrete services which provide the required capabilities. When the abstract service receives a request, it chooses one of these services and invokes it. Which concrete service is chosen depends on the abstract service's implementation. In the simplest case, the abstract service only selects the next concrete service on the list. However, it would be possible to give the abstract service the capability to dynamically assess each concrete service and to choose the one that optimizes the client's QoS requirements. Interface and data incompatibilities are solved by predefined wrappers.

This approach has multiple benefits:

- Usually, a client does not care which specific service handles his requests, as long as the job will be done successfully and in accordance with the contract. The abstract service design supports this notion by providing the capabilities to separate the required abilities from the actual implementation.
- The available list of concrete services enables the abstract service to provide enhanced compensation possibilities.
- The definition of an abstract service can be done independently from the business process in which it will be used. It can therefore be reused in multiple applications. If a specific service implementation is no longer usable, then the business process does not have to be changed, as this is managed in the abstract service.

Figure 1.11 depicts the basic structure of an abstract service. Four interfaces are supplied to the outside: The service operations for which the abstract service has been defined can be accessed via the *abstract service interface*. A contract can be exchanged or negotiated by using the *contract exchange interface*. Execution events of a service (e.g. a failure) can be signaled via the *event interface*. Compensations can be triggered from the outside using the *compensation interface*.

On the inside, the main component is the *management* unit, which receives and processes requests, selects and invokes concrete services, and handles compensations. In order to do so, it has several elements at its disposal:

- *Concrete service list*: Contains the details of all available concrete services.
- *Concrete service wrappers*: Define the mapping of the generic abstract service interface to the specific interface of each concrete service.
- *Request log*: Holds all requests of the current session.
- *Compensation rules repository*: Manages the rules that control the compensation handling process.
- *Contract repository*: Contains the existing contracts with the different clients.

1.4.2 Adapter

Abstract services could be used in conjunction with a wide variety of technologies. Therefore, it would be preferable if the definition of the abstract service itself could be generic. However, the participation in a transaction requires capabilities that are different for each transaction management specification.

That is why the transaction specific requirements are encapsulated in a so-called *adapter* (see Fig. 1.11). An abstract service registers with this adapter, which in turn registers with the transaction coordinator. To the coordinator it appears as if the abstract service itself has registered and sends the status messages. When the abstract service invokes a concrete service, it forwards the information about the adapter, which functions as a coordinator for the service. The service registers accordingly at the adapter as a participant in the transaction.

As can be seen, the adapter works as a mediator between the abstract service, the concrete service, and the transaction coordinator. The adapter receives all status messages from the concrete service and is thus able to process them before they reach the actual coordinator. Normal status messages can be forwarded directly to the coordinator, while failure messages can initiate the internal compensation handling through the abstract service. If the adapter receives such an error message, it informs the abstract service that can then assess the possibility of compensation, which includes checking both the existing compensation rules and the restriction feature model. The adapter will be informed about the decision, and can act accordingly. If for example the replacement of a failed concrete service is possible, then the adapter will deregister this service and wait for the replacement to register. In this case, the failure message will not be forwarded to the transaction coordinator. The compensation assessment could of course also show that a compensation is not possible (or permitted). In such a case, the adapter will simply forward the failure message to the coordinator, which will subsequently initiate the abort of the transaction.

1.4.3 Compensation Protocol

While the compensation rules specify when and how a compensation can be performed, the compensation protocol controls the external compensation process itself and its interaction with the different participants.

An externally triggered compensation always has the purpose of changing one particular request that has already been processed at the service. More specifically, the compensation request contains the original request with its data that has to be changed (`request1(data1)`), and the new request-data (`data2`) to which the original request has to be changed to (`request1(data2)`). The participants in the protocol are the *abstract service*, the *client* which uses the abstract service in its business process, the *initiator* which triggers the external compensation (either the client itself, or any other authorized source like an administrator), the *concrete service* which is currently being utilized by the abstract service, and the *transaction coordinator*. An externally triggered compensation can only be performed if the transaction in which the abstract service participates has not yet finished, as this usually has consequences for the client due to result resending.

The protocol consists of two stages. The first stage is the *compensation assessment*: As soon as the abstract service receives a request for a compensation, it checks whether it is feasible and what the costs would be. To that end, predefined

compensation rules are being used, which consist of a *compensation condition* (defines when a compensation rule can be applied) and a *compensation plan* (defines the compensation actions that have to be performed). The second stage of the protocol is the *compensation execution*, which performs the actual compensation according to the plan. Whether this stage is actually reached depends on the initiator: After the assessment has been completed and has come to a positive conclusion, the initiator, based on this data, has to decide whether the compensation should be performed.

As the client and the initiator of an external compensation can differ, the protocol contains the means to inform the client about the compensation process. It also ensures that the current concrete service and the transaction coordinator are informed about the status of the external compensation, as it is possible that the concrete service's (and thus the abstract service's) state changes due to the external compensation. The concrete service has to enter a specific external compensation handling procedure state for this purpose. While the concrete service is in this state, it will wait for additional requests from the abstract service, and the coordinator is not allowed to complete the transaction. While assessing the possibilities for a compensation, and while performing it, the abstract service cannot process additional requests (and either has to store the requests in a queue, or has to reject them with an appropriate error message).

Because of the requirements of the compensation protocol, it is necessary to adapt the normal transaction protocol with additional state changes regarding the coordinator and participant (i.e. the concrete service). This has been done in our implementation for the `BusinessAgreementWithCoordinatorCompletion` protocol (refer to [11]), using an extended version introduced in [2] as a basis that uses transaction dependency graphs in order to solve cyclic dependencies. The result of the state diagram adaptation for the compensation protocol is depicted in Fig. 1.12.

Two new states have been introduced, `ExCompensation I` and `ExCompensation II`. While both represent the external compensation handling procedure state which the concrete service has to enter, it is necessary to distinct between them, because depending on the former state different consequential transitions exist.

If the concrete service as participant is currently either in the `Active` state or the `Completing` state when receiving an `ExCompensate` notification from the adapter, it will enter the `ExCompensation I` state. While the concrete service is in this state, it will wait for new requests from the abstract service, and the coordinator will not finish the transaction. If the external compensation procedure is canceled after the assessment has been performed, the concrete service will be instructed to re-enter its former state by receiving either an `Active` or a `Complete` instruction from the adapter. The transaction processing can then continue in the normal way. In contrast, if the external compensation is executed and performed successfully, the concrete service will receive an `ExCompensated` message, which instructs it to enter the `Active` state. This is necessary for two reasons: Firstly, because any additional requests as part of the external compensation handling require that the participant again performs the `Completing` operations. And secondly, because the abstract service's client will be informed about the external compensation that

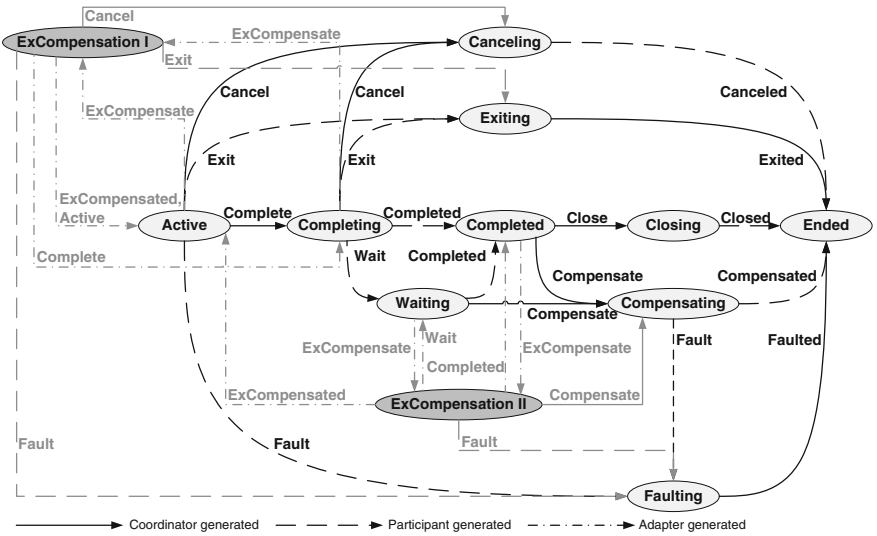

Fig. 1.12 The state diagram of the `BusinessAgreementWithCoordinatorCompletion` protocol with extensions for the external compensation handling

has been performed, and it is possible that additional operations are required by the client as a consequence of the compensation.

In addition to these options within the `ExCompensation I` state, the same transitions exist as in the `Active` and `Completing` states, i.e. the coordinator can `Cancel` the operations, and the participant can `Exit` or send a `Fault` notification.

If the concrete service is either in the `Waiting` or `Completed` state when receiving an `ExCompensate` message, it will enter the `ExCompensation II` state. In principle, the state has the same meaning as `ExCompensation I`: The concrete service will wait for new abstract service requests, and at the same time the coordinator is not allowed to finish the transaction. The concrete service will be notified to enter the `Active` state through an `ExCompensated` message after a successful external compensation execution. However, in contrast to `ExCompensation I`, different consequential transitions are available, and therefore it is necessary to separate these two states. In case of a compensation abort, the concrete service can be instructed to re-renter its former state through a `Wait` or `Completed` message. Moreover, a `Fault` message can be sent to signal an internal failure. Finally, the coordinator can send a `Compensate` instruction while the concrete service is in the `ExCompensation II` state. The concrete service can only be instructed to `Compensate` if it is either in the `Waiting` or the `Completed` state. Therefore, it is necessary to introduce `ExCompensation II`, as this option is not available for the `Active` and `Completing` states and thus may not be permitted within `ExCompensation I`.

The extended state diagram contains new transitions generated by the adapter in addition to the ones from the participant (i.e. the concrete service) and the coordinator.

This is actually a simplification, because although the adapter creates the messages and sends them to the coordinator and the participant, both are not aware of the fact that the adapter has sent them. To the coordinator it always looks as if the participant has sent the messages, while the participant thinks that the coordinator has sent them, as both are unaware of the extended transaction environment. Therefore, in order to obtain a state diagram that shows only transitions generated by either the coordinator or the participant, it would be necessary to create two different state diagrams, one from the participant's view and one from the coordinator's.

1.4.4 Application on the Client and Provider Side

The abstract service design can be applied on both, the client and the provider side. A client who wants to create a new distributed application using services provided by multiple providers can utilize abstract services in two different ways:

1. The client can include the abstract service from a provider in his new business process, and can use the added capabilities.
2. The client can define a new abstract service, which manages multiple concrete services that can perform the same task.

The main goal of a Web service provider is a successful and stable execution of the client's requests in accordance with the contracts. If the service of a provider fails too often, he might face contractual penalties, or the client might change the provider. He can use abstract services in order to enhance the reliability and capability of his services by creating an abstract service which encapsulates multiple instances or versions of the same service. These can be used in case of errors to compensate the failure without the need for a transaction abort.

1.4.5 Client Contracts

While the different compensation capabilities of an abstract service allow the handling of internal and external compensations, it may not always be desirable for a client that these functionalities are applied. The abstract service environment therefore allows the definition and evaluation of *contracts*.

A client will negotiate a contract with the abstract service before sending the first request. This contract not only contains legal information and the Service Level Agreement, but can also specify (using a restriction feature model as described in Sect. 1.2.5) which compensation operations the abstract service is permitted to apply. The abstract service adapts dynamically to this contract by checking the restrictions defined in it prior to performing a compensation: A compensation rule may only be applied if all necessary compensation operations are permitted via the contract. It can thus happen that although a compensation rule exists for handling a compensation, the abstract service will not apply it because the contract restricts the use of required

compensation operations. Accordingly, an abstract service that is not allowed to use any compensation capabilities will act exactly like a standard Web service. A client therefore *can* make use of the forward recovery capabilities, but he does not *have* to, and thus always has the control over the environment's forward recovery compensation handling features.

Because of this ability to dynamically adapt to each client's contract, it is possible to use the same abstract service in a wide variety of distributed applications with differing requirements regarding compensation handling.

1.4.6 Transaction Environment Adaptation

The abstract service and adapter approach has been designed as an extension of the current transaction coordination structure so that it is easy to integrate it into existing environments and different transaction protocols. Therefore, it is not necessary to change either the client, coordinator or concrete service in order to use the internal compensation handling capability: An abstract service that manages different concrete services and that is able to replace failed concrete services can be used like a normal Web service and without any changes to the transaction protocol.

However, the introduced external compensation functionality for changing already processed requests requires some changes in the transaction environment:

1. It is necessary to extend the existing transaction specification protocols to provide the capability to perform external compensations. This has been shown for the `BusinessAgreementWithCoordinatorCompletion` protocol in Sect. 1.4.3. Accordingly, the coordinator and the participating concrete service have to be able to handle this adapted protocol.
2. The external compensation process requires that reports about a performed compensation or the resending of results can be sent to the client of a transaction. It is therefore necessary that the client provides the expected interfaces and that he is able to process these reports in accordance with his business process.

The extent of the changes thus depends on the compensation requirements.

1.4.7 Middleware Prototype

The described design approach has been implemented as a prototype in order to verify the design and the protocols. The implementation has been done using Apache Tomcat as Web container, and Apache Axis as SOAP engine. The WS-Transaction specification has been chosen for the transaction coordination, more specifically the adapted `BusinessAgreementWithCoordinatorCompletion` protocol that has been introduced in Sect. 1.4.3. The implementation has been published online at SourceForge.net as the *FROGS* (**f**orward **r**ecovery c**o**mpensation handlin**g** **s**ystem) project: http://sourceforge.net/projects/frogs/.

1.5 Discussion

The evaluation of the approach is discussed in detail in [19]. Here we provide a summary of the findings: The experiments showed that in our environment about twice as many transactions as in a standard environment finish successfully. Furthermore, a similar improvement can also be found if we look at how many transactions finish in one minute. The number of messages sent in the environment is of course higher, which is, however, compensated by the increased number of transactions that do not have to roll back. Also, the number of additional messages is justified well enough if the overall cost of forward recovery is lower than the cost of the rollback and cumulative cascading rollbacks.

The current approach still has some shortcomings. The transition from design to compensation rules needs to be studied in order to support it via semi-automatic tools. Also, the algorithms for matching capability and requirement models require further studies, as the proposed algorithm is limited to an exact match. Especially approximation and similarity methods can be beneficial in this context. In addition, the support for different types of configuration models seems quite useful to study. So far, we have concentrated our efforts on defining the architecture and the required protocol. It will be necessary to do a further analysis of the proposed protocol to ensure that it is complete and is not susceptible to race conditions, which can occur in a real-life environment where less than optimal conditions exist, messages can be delayed or lost, and many concurrent accesses can exist.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services - Concepts, Architectures and Applications. Springer (2003)
2. Alrifai, M., Dolog, P., Nejdl, W.: Transactions Concurrency Control in Web Service Environment. In: ECOWS '06: Proceedings of the European Conference on Web Services, pp. 109–118. IEEE, Washington, DC, USA (2006). DOI 10.1109/ECOWS.2006.37
3. Choi, S., Jang, H., Kim, H., Kim, J., Kim, S.M., Song, J., Lee, Y.J.: Maintaining Consistency Under Isolation Relaxation of Web Services Transactions. In: A.H.H. Ngu, M. Kitsuregawa, E.J. Neuhold, J.Y. Chung, Q.Z. Sheng (eds.) WISE, *Lecture Notes in Computer Science*, vol. 3806, pp. 245–257. Springer (2005)
4. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language (WSDL) 1.1. W3C note, W3C (2001)
5. Dolog, P., Nejdl, W.: Using UML-Based Feature Models and UML Collaboration Diagrams to Information Modelling for Web-Based Applications. In: T. Baar, A. Strohmeier, A. Moreira, S.J. Mellor (eds.) Proc. of UML 2004 — The Unified Modeling Language. Model Languages and Applications. 7th International Conference, *LNCS*, vol. 3273, pp. 425–439. Springer (2004)
6. Dostal, W., Jeckle, M., Melzer, I., Zengler, B.: Service-orientierte Architekturen mit Web Services. Spektrum-Akademischer Verlag (2005)
7. Gray, J.: The Transaction Concept: Virtues and Limitations. In: VLDB 1981: Intl. Conference on Very Large Data Bases, pp. 144–154. Cannes, France (1981)

8. Greenfield, P., Fekete, A., Jang, J., Kuo, D.: Compensation is Not Enough. In: 7th International Enterprise Distributed Object Computing Conference (EDOC 2003), pp. 232–239. IEEE Computer Society, Brisbane, Australia (2003)
9. Greenfield, P., Fekete, A., Jang, J., Kuo, D., Nepal, S.: Isolation Support for Service-based Applications: A Position Paper. In: CIDR, pp. 314–323 (2007)
10. Greenfield, P., Kuo, D., Nepal, S., Fekete, A.: Consistency for Web Services Applications. In: Proceedings of the 31st international conference on Very large data bases, VLDB '05, pp. 1199–1203. VLDB Endowment (2005). URL http://dl.acm.org/citation.cfm?id=1083592.1083731
11. Ltd., A.T., Systems, B., Ltd., H., Corporation, I., Technologies, I., Corporation, M.: Web Services Business Activity Framework (2005). Published at ftp://www6.software.ibm.com/software/developer/library/WS-BusinessActivity.pdf
12. Ltd., A.T., Systems, B., Ltd., H., Corporation, I.B.M., Technologies, I., Corporation, M.: Web Services Coordination (2005). Published online at ftp://www6.software.ibm.com/software/developer/library/WS-Coordination.pdf
13. Ltd., A.T., Systems, B., Ltd., H., Corporation, I.B.M., Technologies, I., Inc., M.C.: Web Services Atomic Transaction (2005). Published at ftp://www6.software.ibm.com/software/developer/library/WS-AtomicTransaction.pdf
14. Meyer, B.: Applying "Design by Contract". IEEE Computer **25**(10), 40–51 (1992)
15. Nielsen, H.F., Mendelsohn, N., Moreau, J.J., Gudgin, M., Hadley, M.: SOAP Version 1.2 Part 1: Messaging Framework. W3C recommendation, W3C (2003)
16. Pullum, L.L.: Software Fault Tolerance — Techniques and Implementation. Artech House, Inc., Norwood, MA, USA (2001)
17. Schäfer, M., Dolog, P.: Feature-Based Engineering of Compensations in Web Service Environment. In: M. Gaedke, M. Grossniklaus, O. Díaz (eds.) Web Engineering, 9th International Conference, ICWE 2009, *Lecture Notes in Computer Science*, vol. 5648, pp. 197–204. Springer, San Sebastián, Spain (2009)
18. Schäfer, M., Dolog, P., Nejdl, W.: Engineering Compensations in Web Service Environment. In: P. Fraternali, L. Baresi, G.J. Houben (eds.) ICWE2007: International Conference on Web Engineering, *LNCS*, vol. 4607, pp. 32–46. Springer Verlag, Como, Italy (2007)
19. Schäfer, M., Dolog, P., Nejdl, W.: Environment for Flexible Advanced Compensations of Web Service Transactions. ACM Transactions on Web **2**(2) (2008)
20. Yang, Z., Liu, C.: Implementing a Flexible Compensation Mechanism for Business Processes in Web Service Environment. In: ICWS '06. Intl. Conference on Web Services, pp. 753–760. IEEE Press, Salt Lake City, Utah, USA (2006)