Engineering Compensations in Web Service Environment

Michael Schäfer[1], Peter Dolog[2], and Wolfgang Nejdl[1]

[1] L3S Research Center, University of Hannover,
Appelstr. 9a, D-30167 Hannover, Germany,
Michael.Schaefer@stud.uni-hannover.de, nejdl@l3s.de
[2] Aalborg University, Department of Computer Science,
Fredrik Bajers Vej 7E,DK-9220 Aalborg East, Denmark
dolog@cs.aau.dk

Abstract. Business to business integration has recently been performed by employing Web service environments. Moreover, such environments are being provided by major players on the technology markets. Those environments are based on open specifications for transaction coordination. When a failure in such an environment occurs, a compensation can be initiated to recover from the failure. However, current environments have only limited capabilities for compensations, and are usually based on backward recovery. In this paper, we introduce an engineering approach and an environment to deal with advanced compensations based on forward recovery principles. We extend the existing Web service transaction coordination architecture and infrastructure in order to support flexible compensation operations. A contract-based approach is being used, which allows the specification of permitted compensations at runtime. We introduce the *abstract service* and *adapter* components which allow us to separate the compensation logic from the coordination logic. In this way, we can easily plug in or plug out different compensation strategies based on a specification language defined on top of basic compensation activities and complex compensation types. Experiments with our approach and environment show that such an approach to compensation is feasible and beneficial.

1 Introduction

The Web service environment has become the standard for Web applications supporting business to business transactions and user services. Processes such as payroll management or supply chain management are realized through Web services. In order to ensure that the results of the business transactions are consistent and valid, Web service coordination and transaction specifications [12, 13, 11] have been proposed. They provide the architecture and protocols that are required for transaction coordination of Web services.

The transaction compensation [7] is a replacement for an operation that was invoked but failed for some reason. The operation which replaces the original one either undoes the results of the original operation, or provides similar capabilities

as the original one. The notion of compensation was introduced for environments where the isolation property of transactions is relaxed but the atomicity needs to be maintained. Several protocols have been proposed to control transactional processes with compensations [20].

Current open specifications for transaction management in Web service environment provide only limited compensation capabilities [8]. In most cases, the handling of a service failure is restricted to *backward recovery* in order to maintain consistency, i.e. all running services are aborted, and all already performed operations are reversed [1]. This approach is very inflexible and can result in the abortion of many services and transactions. Especially if dependencies between multiple transactions exist, the failure of one service can lead to cascading compensations. Furthermore, 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.

In this paper, we investigate an engineering approach for advanced compensation operations adopting *forward recovery* within Web service transactions. Forward recovery proactively changes the state and structure of a transaction after a service failure occurred, and thus enables the transaction to finish successfully. The main idea is the introduction of a new component called an *abstract service*, which functions as a mediator for compensations, and thus hides the logic behind the introduced compensations. Moreover, it specifies and manages potential replacements for primary Web services to be used within a transaction. The compensations are performed according to predefined rules, and are subject to contracts [14]. We introduce a framework based on the abstract services, which enables the compensations described in the compensation specifications.

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 the flexibility and efficiency.
- The environment can handle both, 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, and consistency is achieved through well defined communication protocols.
- 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 paper is structured as follows. Section 2 introduces the motivating scenario, which will be used in the paper in order to exemplify the

concepts. Section 3 introduces the proposed design for an infrastructure that is able to handle internally and externally triggered compensations without transaction aborts, and describes the basic components and compensation specifications based on compensation activities and compensation types. A prototype implementation of the design is described in section 4, along with two experiments that use the new compensation capabilities. Section 5 reviews related work in the area of forward recovery. Section 6 concludes this paper and provides a direction for future work on this topic.

2 Motivating Scenario

The motivating scenario for this paper is a company's monthly payroll processing. In order to introduce real-life dependencies, both, the company's and the employee's responsibilities are considered.

Company: In the first step of the payroll processing procedure, 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. First of all, the salary is transferred from the company's account to the employee's account. Then the company transfers the employee's income tax to the account of the fiscal authorities. Finally, the company prints the payslip and sends it to the employee.

Employee: The employee has only one task which he has to perform each month in this scenario: He transfers the monthly instalment 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 Web services from multiple providers. The two business processes use transactions in order to guarantee a consistent execution of all required operations. This is depicted in Figure 1. Only the services of transaction T1 are shown.

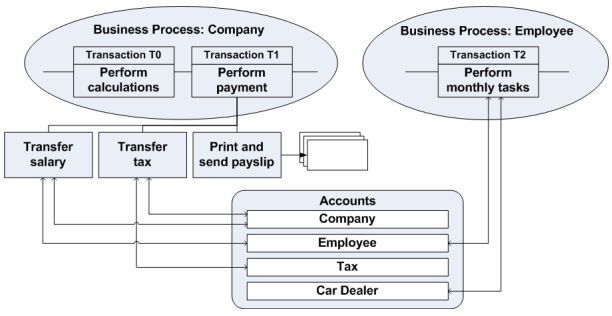

Fig. 1. The motivating scenario

It is obvious that there are multiple dependencies in this simple scenario, between and within these transactions. Therefore, it is vitally important that no

transactions have to be aborted and compensated in order to avoid cascading compensations. However, such a situation can become necessary quite easily:

1. It can always happen that a service which participates in a transaction fails. Here, it could be that the service that handles the transfer of the salary fails due to an internal error. The transaction inevitably has to be aborted, even though the error might be easily compensatable by using a different service that can perform the same operation. Such a *replacement* is encouraged by the fact that usually multiple services exist that have the same capabilities.
2. A mistake has been made regarding the input data of an operation. In this scenario, it could be that the calculation of the salary is inaccurate, and too much has been transferred to the employee's account. The flaw is spotted by an administrator, but the only option is again to abort the complete transaction, although it would be very easy to correct the mistake by transferring the sum that has been paid too much back to the company's account.

Although it should be possible to handle these situations without the need to cancel and compensate the transaction(s), current technology does not allow to do so in a sensible way.

3 Web Service Environment with Transaction Coordination

We base our work on Web service coordination and transaction specifications [12, 13, 11]. These transaction specifications provide a conceptual model and architecture for environments where business activities performed by Web services are embedded into transactional contexts.

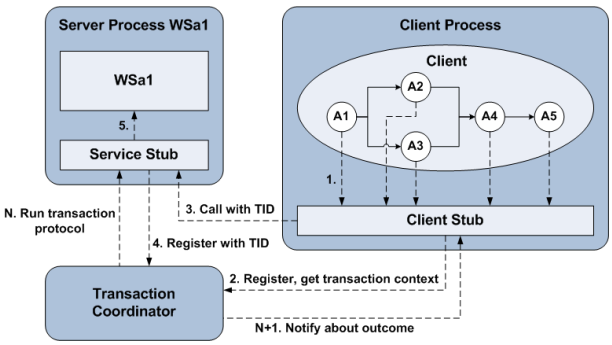

Fig. 2. Transactional environment for Web services adopted from [1]

Figure 2 depicts an excerpt of such an environment with the main components. The client runs business activities A1 to A5, which are embedded in a

transactional context. The transactional context and conversation 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 transaction coordinator is then responsible for running appropriate protocols, such as two phase commit or some of the distributed protocols for Web service environments such as [2].

As pointed out above, the compensation capabilities are left to the client business activities according to the specifications in [12, 13, 11]. We extend the architecture and the infrastructure based on those specifications, so that it can handle internally and externally triggered compensations. Figure 3 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, transaction coordinators and Web service providers operate. Clients, instead of invoking concrete Web services, invoke abstract services which wrap several services and compensations for them. The adapter functions as a mediator between transaction coordinator, abstract service and concrete service to ensure proper transactional context.

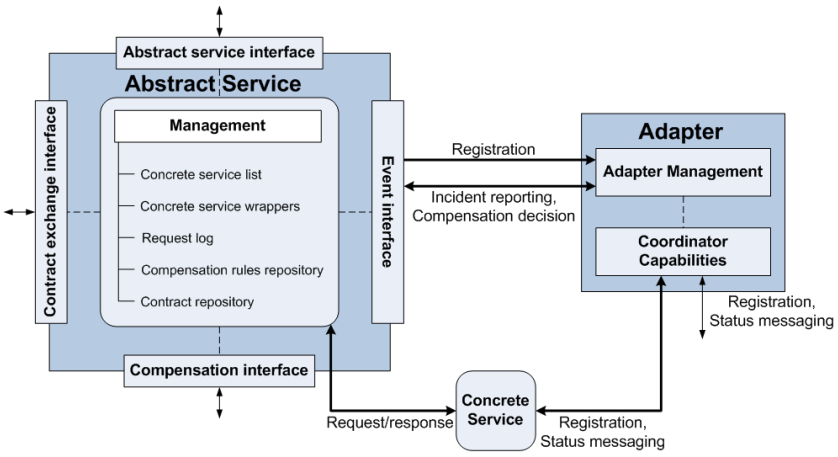

Fig. 3. The abstract service and adapter

3.1 Abstract Service

The central element of the extension is the notion of an *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.

Distributed applications consisting of collaborating Web services have the advantage that normally single operations can be performed by multiple services from different providers. Which service will be chosen depends usually on the quality of service (QoS) requirements of the distributed application. The abstract service takes advantages of the existing diversity. 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 Web services (called *concrete services*) which provide the required capabilities. When the abstract service receives a request, it chooses one of these services and invokes it. Interface and data incompatibilities between the abstract interface and the interfaces of the concrete services 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 without the need for changes. If a specific service implementation is no longer usable, then the business process does not have to be changed, as this is being managed in the abstract service.

Figure 3 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.

3.2 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 Figure 3). An abstract service registers at this adapter, which in turn registers with the transaction coordinator. To the coordinator it looks 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 it 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 the 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, which can then assess the possibility of compensation. The adapter will then 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 desirable). In such a case, the adapter will simply forward the failure message to the coordinator, which will subsequently initiate the abort of the transaction.

3.3 Compensation Specifications

Compensation specifications enable the abstract service to handle both kinds of compensations: Internally triggered compensations (arising from internal errors) and externally triggered compensations. An example for an externally triggered compensation could be the handling of the mistake spotted by an administrator as described in the motivation scenario section. We distinguish between *compensation activities* and *compensation types* in our compensation specifications, whose interaction are shown in Figure 4.

Basic Compensation Activities are the basic operations which can be used in a compensation. *ServiceReplacement* replaces the currently used Web service with a different one, which can offer the same capabilities and can thus act as a replacement. *LastRequestRepetition* resends the last request to the Web service. *PartialRequestRepetition* resends the last n requests from the request sequence of the current session (i.e. within the current transaction) to the Web service, while *AllRequestRepetition* resends all requests. *CompensationForwarding* forwards the

Nr.		Compensation Type	Compensation Activities: ServiceReplacement	LastRequestRepetition	PartialRequestRepetition	AllRequestRepetition	CompensationForwarding	AdditionalServiceInvocation	AdditionalRequestGeneration	ServiceAbortInitiation	RequestSequenceChange	ResultResending
01		NoCompensation										
02	Internal	Repetition		X								
03	Internal	Repetition			X							X
04	Internal	Replacement	X	X								
05	Internal	Replacement	X		X							X
06	Internal	Replacement	X			X						X
07	External	Forwarding	(X)	(X)	(X)	(X)	X	(X)	(X)	(X)	(X)	(X)
08	External	AdditionalService						X				
09	External	AdditionalRequest							X			
10	External	SessionRestart				X				X	X	X

X Included compensation activity (X) Possibly included compensation activity

Fig. 4. The compensation types and their included activities

external compensation request to a different component, which will handle it. *AdditionalServiceInvocation* invokes an additional (external or internal) service, which performs some operation that is important for the compensation (e.g. the invocation of a logging service, which collects data about a specific kind of compensation). *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 Web 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.

Compensation Types aggregate multiple compensation activities, and thus form complex compensation operations, as shown in Figure 4. These types are the compensation actions which can be used for internal and external compensations, and which form the basis of the compensation specification language. There are currently 7 different compensation types.

The most simple type is *NoCompensation*, which does not perform any operation. If a Web service fails, then this will be signaled to the transaction coordinator, which will initiate the transaction abort.

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 resend to a Web 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, which has to be completely repeated after the failure of the final request. 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 Web 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 requeststo the concrete 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 can not 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 resend. The requested change will be realised by a change in the request sequence prior to the resending.

Compensation Protocol controls the compensation process 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), 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 it 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 de-

pends 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 or not.

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 transaction coordinator is informed about the status of the external compensation, because the assessment and the execution stages have consequences for the abstract service's status in the transaction. While assessing the possibilities for a compensation, and while performing it, the abstract service can not process additional requests (and either has to store the requests in a queue, or has to reject them with an according error message). Moreover, its status can change as a result of a successful compensation.

3.4 Application on the Client and Provider Side

The abstract service design can be applied on both, the client and the provider side. A client which 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 its 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.

4 Discussion and Experiments

The described design approach has been used in a prototype implementation based on the scenario in section 2, and we performed two experiments with the implemented environment.

The four services participating in the payment transaction have been realized as abstract services. The abstract services manage the standard Web services performing the required operations as concrete services. 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 BusinessAgreementWithCoordinatorCompletion protocol with the extension for transaction concurrency control that has been introduced in [2]. It is necessary for externally triggered compensations

that the transaction coordinator is able to adapt to the changes that have to be performed in the process.

The first experiment was devoted to the evaluation of the compensation of an internal service error. In this case, a failure of the concrete service on the provider side is simulated. Figure 5 shows the setup for the *transfer salary* operation: The abstract service *AS1* on the client side currently uses a concrete service that is itself an abstract service (*AS2*), which is operated by a service provider. The abstract service *AS2* uses *Web Service 1*, which performs the required operations. Figure 5 also depicts the interconnection of the services: *AS1* is registered as a participant at the *Transaction Coordinator* via *Adapter 1*, *AS2* is registered at *Adapter 1* via *Adapter 2*, and *Web Service 1* is registered at *Adapter 2*.

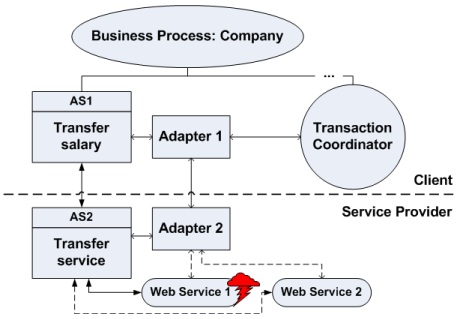

Fig. 5. Compensation on the provider side

Now *Web Service 1* fails due to an internal error, and is thus not able to perform all operations required for the salary transfer. Instead of informing the transaction coordinator, abstract service *AS2* is informed, which assesses its compensation rules, the contract, and the available substitution services, and decides that a compensation is possible. *Web Service 1* is discarded and the request that failed is send to *Web Service 2*, which registers at the adapter. *Web Service 2* is another instance of the same service, and can therefore simply continue with the request as it operates on the same local resources. This scenario shows that the signal of the service failure can be intercepted and the service replaced, without the need to cancel the complete transaction.

The second experiment evaluates an externally triggered compensation. Figure 1 summarizes the operations on the different accounts in the scenario described in section 2. In this experiment, an administrator has found an error in the calculation of the salary: The company transferred 50 units too much to the account of the employee. The administrator directly sends a compensation request to the abstract service that handles the salary transfer (*AS1*). The abstract service assesses the request by consulting its compensation rules. In this scenario, the rules specify that this compensation is only allowed if the

employee's account would still be in credit after the additional debit operation, in order to avoid the employee's account being in debit after the transaction.

Nr.	Transaction	Company (C)	Employee (E)	Tax (T)	Car Dealer (D)
		10.000	0	Y	Z
01	T1.debit(C,1.000)	9.000			
02	T1.credit(E,1.000)		1.000		
03	T1.debit(C,500)	8.500			
04	T1.credit(T,500)			Y+500	
05	T2.debit(E,150)		850		
06	T2.credit(D,150)				Z+150
		8.500	850	Y+500	Z+150

Table 1. The transfer operations on the accounts in the scenario

The result of the assessment is positive, which is reported to the administrator, who can decide based on this data whether the compensation should be performed. He decides that the compensation is necessary. The abstract service compensates operations 01 and 02 from Table 1 by creating an additional debit and credit operation, as can be seen in Table 2. The operations transfer 50 from the employee's account back to the company's account, which thus compensates the initial problem. As an additional service, the abstract service initiates a precautionary phone call, which informs the employee about the change.

Nr.	Transaction	Company (C)	Employee (E)	Tax (T)	Car Dealer (D)
...	...	...	...	...	...
07	**T1.debit(E,50)**		**800**		
08	**T1.credit(C,50)**	**8.550**			
		8.550	**800**	Y+500	Z+150

Table 2. The additional operations on the accounts

Subsequently, the compensation will be reported to the client, who has to assess whether any other services are affected according to its business process. It decides that the tax transfer does not have to be changed, while the payslip has to be updated, as the details of the salary have changed. The business process therefore initiates a compensation on the respective service, which handles this request by printing and mailing a new payslip. This shows that even the more complex initial problem could be solved without the need to abort the transaction.

These two experiments have shown that the proposed design is successful in employing flexible compensation strategies in Web service transactions. It is thus possible to develop more robust distributed applications, where the abstract

services are able to adapt their compensation rules to the contract they have with the client. Especially in long-running transactions, this approach helps to avoid unnecessary transaction aborts, and therefore saves money and time. While it is of course still possible that the abstract service itself encounters an error, it at least provides the capabilities to avoid transaction aborts due to concrete service failures. Moreover, it is possible to mix the new design with existing technology: The new capabilities can be used, but do not have to be, as an abstract service can be employed like any other normal Web service.

However, the new functionality of the design with its advanced compensation abilities has its costs. By introducing additional components like the abstract services and the adapters, the overall structure of a distributed application becomes more complex. The outsourcing of compensation logic to the abstract services simplifies the business process definitions, but at the same time the distributed compensation logic can make maintenance more difficult. And finally, the added components require additional messaging, and therefore the design increases the total number of messages that have to be sent.

The current implementation is a proof-of-concept of the proposed design architecture, and is still limited regarding certain aspects. The prototype of the abstract service uses only synchronous requests and does not allow parallel requests. Nevertheless, the same principles can be applied in this case, although additional request queue management will be required. Accordingly, the execution of compensation actions is currently performed only sequentially.

5 Related Work

Forward recovery can be realized by using dynamic workflow changes, as described in [17, 19], which allow the semi-automatic adaptation of a workflow in case of errors. A change of the workflow process can for example consist of a deletion or jump instruction, or the insertion of a whole new process segment. The change can either be done on a running instance, or it can be performed on the scheme which controls the workflow, and which results in a change in all running instances. Refer to [18] for details. Although this approach is very powerful, it has two major disadvantages. Firstly, it is in most cases only possible to perform these adaptations semi-automatically. Changing a workflow requires a lot of knowledge about the process and the current state it is in, and the implications a change would have. Therefore, it is often necessary for a human administrator to specify and control the change. Secondly, these kinds of workflow changes require a very strict definition of the process, including for example data and control links. Ad-hoc changes of business processes with normal orchestration languages like WS-BPEL (see [6]) is very difficult [9]. [5] provides a mechanism to overcome this difficulty through a compensation handler. Our approach provides a more flexible solution for compensations orthogonal to the business processes, concrete services, and transaction coordination.

Our compensation approach can be used with the Enterprise Service Bus (ESB) [3], a powerful messaging infrastructure for business to business inte-

gration with Web services. The abstract service and adapter can be integrated through the ESB flexible extension mechanism. In this way, ESB can serve as a platform to exchange extended messages between business process, abstract services and adapters involved in the compensation conversation. Our approach can be used independently of ESB, employing ESB on top of the introduced infrastructure to integrate abstract services with workflow activities.

[16] introduces a notion of compensable Web services by specifying operations which can revert the execution. In our approach, we allow for a more complex specification of forward recovery compensations, which can be introduced at the client side, mediator side, as well as provider side. Two related approaches to a flexible compensation mechanism for business processes are proposed in [20, 10]. In both cases, the focus is put on backward recovery. The compensation logic is treated as a part of coordination logic. In our approach, we separate the coordination from the compensation logic to provide for more flexibility.

6 Conclusions and Further Work

We have described a new design approach for complex compensation strategies in current transaction standards. Two new components have been described, the *abstract service*, which manages replacement services and compensation rules, and the *adapter*, which separates the coordination protocol specific functions from the generic definition of the abstract service. We have also presented the protocol that handles the assessment and processing of externally triggered compensations. The design and the protocol have been successfully validated in a prototype implementation.

Regarding future work, we plan to run additional experiments with different compensation scenarios. Moreover, it will be necessary to further analyze the impact of the new compensation capabilities on the business process definitions. At the moment, it is only assumed that the business process is able to adapt to the signaled compensations. It will be required to analyze possible extensions of existing orchestration languages like BPEL in order to include the new capabilities. The current implementation will be extended to support the management of parallel request processing, and the definition of compensation rules will be adapted accordingly.

References

1. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services - Concepts, Architectures and A pplications.* Springer, November 2003.
2. M. Alrifai, P. Dolog, and W. Nejdl. Transactions Concurrency Control in Web Service E nvironment. In *ECOWS '06: Proceedings of the European Conference on Web Services*, pages 109–118, Zurich, Switzerland, December 2006. IEEE.
3. D. A. Chappell. *Enterprise Service Bus.* O'Reilly Media, Inc., 2004.
4. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3C note, W3C, March 2001.

5. G. Dobson. Using ws-bpel to implement software fault tolerance for web services. In *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 126–133, Washington, DC, USA, 2006. IEEE Computer Society.
6. A. Alves et al. Web Services Business Process Execution L anguage Version 2.0, 2007. Published online at http://docs.oasis- open.org/wsbpel/2.0/CS01/wsbpel-v2.0-CS01.pdf.
7. J. Gray. The transaction concept: Virtues and limitations. In *VLDB 1981: Intl. Conference on Very Large Data Bases*, 1981.
8. P. Greenfield, A. Fekete, J. Jang, and D. Kuo. Compensation is not enough. In *7th International Enterprise Distributed Object Computing Conference (EDOC 2003)*, pages 232–239, Brisbane, Australia, September 2003. IEEE Computer Society.
9. D. Karastoyanova, A. Houspanossian, M. Cilia, F. Leymann, and A. P. Buchmann. Extending bpel for run time adaptability. In *Ninth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005)*, Enschede, The Netherlands.
10. L. Lin and F. Liu. Compensation with dependency in web services composition. In *International Conference on Next Generation Web Services Practices (NWeSP 2005)*, pages 183–188, Seoul, KOREA, August 2005. IEEE Press.
11. Arjuna Technologies Ltd., BEA Systems, Hitachi Ltd., IBM Corporation, IONA Technologies, and Microsoft Corporation. Web Services Business Activity Framework, 2005. Published online at ftp://www6.software.ibm.com/software/developer/library/WS- BusinessActivity.pdf.
12. Arjuna Technologies Ltd., BEA Systems, Hitachi Ltd., International Business Machines Corporation, IONA Technologies, and Microsoft Corporation. Web Services Coordination, 2005. Published online at ftp://www6.software.ibm.com/software/developer/library/WS- Coordination.pdf.
13. Arjuna Technologies Ltd., BEA Systems, Hitachi Ltd., International Business Machines Corporation, IONA Technologies, and Microsoft Corporation Inc. Web Services Atomic Transaction, 2005. Published online at ftp://www6.software.ibm.com/software/developer/library/WS- AtomicTransaction.pdf.
14. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.
15. H. F. Nielsen, N. Mendelsohn, J. J. Moreau, M. Gudgin, and M. Hadley. SOAP version 1.2 part 1: Messaging framework. W3C recommendation, W3C, June 2003.
16. P. F. Pires, M. R.F. Benevides, and M. Mattoso. Building reliable web services compositions. In *Web, Web-Services, and Database Systems: NODe 2002, Web- and Database-Related Workshops, Erfurt, Germany, October 7-10, 2002. Revised Papers*, Enschede, The Netherlands.
17. M. Reichert and P. Dadam. ADEPTflex: Supporting Dynamic Changes of W orkflow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.
18. M. Reichert, S. Rinderle, U. Kreher, and P. Dadam. Adaptive Process Management with ADEPT2. In *ICDE*, pages 1113–1114. IEEE, 2005.
19. S. Rinderle, S. Bassil, and M. Reichert. A Framework for Semantic Recovery Strategies in Case of Process Activity Failures. In Y. Manolopoulos, J. Filipe, P. Constantopoulos, and J. Cordeiro, editors, *ICEIS*, pages 136–143, 2006.
20. Z. Yang and C. Liu. Implementing a flexible compensation mechanism for business processes in web service environment. In *ICWS '06. Intl. Conference on Web Services*, 2006.