Date of acceptance Grade

Instructor

Recovery Management of Long Running eBusiness Transactions

Minna Ulmala

Helsinki April 9, 2012

UNIVERSITY OF HELSINKI

Department of Computer Science

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta/Osasto – Fakultet/Sektion – Faculty/Section	Laitos – Institution – Department	
Faculty of Science	Department of Computer Science	
Tekijä – Författare – Author		
Minna Ulmala		
Työn nimi – Arbetets titel – Title		
Recovery Management of Long Running eBusiness Transactions		
Oppiaine – Läroämne – Subject		
Computer Science		
Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
Pro Gradu	April 2012	79

Tiivistelmä – Referat – Abstract

eBusiness collaboration and an eBusiness process are introduced as a context of a long running eBusiness transaction. The nature of the eBusiness collaboration sets requirements for the long running transactions. The ACID properties of the classical database transaction must be relaxed for the eBusiness transaction. Many techniques have been developed to take care of the execution of the long running business transactions such as the classical Saga and a business transaction model (BTM) of the business transaction framework. Those classic techniques cannot adequately take into account the recovery needs of the long running eBusiness transactions and they need to be further improved and developed.

The expectations for a new service composition and recovery model are defined and described. The DeltaGrid service composition and recovery model (DGM) and the Constraint rules-based recovery mechanism (CM) are introduced as examples of the new model. The classic models and the new models are compared to each other and it is analysed how the models answer to the expectations.

Neither new model uses the unaccustomed classification of atomicity even if the BTM includes the unaccustomed classifying of atomicity. A recovery model of the new models has improved the ability to take into account the data and control dependencies in the backward recovery. The new models present two different kinds of strategies to recover a failed service. The strategy of the CM increases the flexibility and
the efficiency compared to the Saga or the BTF. The DGM defines characteristics that the CM does not have: a Delta-Enabled rollback, mechanisms for a pre-commit recoverability and for a post-commit recoverability and extends the concepts of a shallow compensation and a deep compensation. The use of them guarantees that an eBusiness process recovers always in a consistent state which is something the Saga, the BTM and the CM could not proof. The DGM offers also the algorithms of the important mechanisms.

ACM Computing Classification System (CCS):
C.2.4 [Distributed Systems]: Distributed applications

Avainsanat – Nyckelord – Keywords
recovery, data dependencies, service composition

Säilytyspaikka – Förvaringställe – Where deposited
Helsingin yliopiston kirjasto Kumpulan kampuskirjasto

Muita tietoja – Övriga uppgifter – Additional information

Contents

1 Introduction

Organizations purchase services from their partners because they want to focus themselves on their core business. eBusiness methods enable an organization to link the purchased services to its business processes and data processing systems enabling the company to reach its business goals. An eBusiness process of an organization usually consists of several services of which some may be implemented as Web services.

An eBusiness process utilizes database transaction concepts [SH05] for safeguarding. However many features differentiate an eBusiness transaction from a classical database transaction. Those features as well the additional requirements of an execution of an eBusiness transaction set expectations for an eBusiness transaction. The ACID properties of the classical database transaction must be relaxed for an eBusiness transaction.

Many techniques have been developed to take care of the execution of the long running business transactions such as the classical Saga [GS87] model and a business transaction model of the business transaction framework [PAP03]. Those classic techniques cannot adequately take into account the recovery needs of the long running eBusiness transactions and they need to be further improved and developed [PAP03][SH05]. The research question of this paper is: how to secure a long running eBusiness transaction to a consistent state through recovery during an eBusiness transaction.

To be able to define how the classic techniques should be improved to better take into account the recovery needs of a long running eBusiness transaction to reach a consistent state the expectations for a new service composition and recovery model are defined. The DeltaGrid service composition and recovery model [UX09] and the Constraint rules-based recovery mechanism [CZM10] are introduced as examples of the new service composition and recovery model. The classic models and the new models are compared to each other and it is analysed how the models answer to the expectations.

In the following, Section 2 introduces two classic techniques to take care of the recovery of the long running eBusiness transactions and defines and describes the expectations for a service composition and recovery model. Section 3 explores the DeltaGrid service composition and recovery model and Section 4 explores the Constraint rules-based recovery mechanism. Section 5 compares the classic models and the new models to each other and analyses how the models answer to the expectations.

2 eBusiness Transactions – Features and Execution Requirements

eBusiness collaboration and an eBusiness process are introduced as a context of the long running eBusiness transaction (subsection 2.1). An eBusiness process is compared to a database transaction and compensation, contingency and an eBusiness transaction are defined (subsection 2.2). ACID-properties are rehearsed (subsection 2.3.1). The features that differentiate an eBusiness transaction from a classical database transaction are introduced as well the additional requirements they set to an execution of an eBusiness transaction compared to the classical database transaction (subsection 2.3.2). The relaxed ACID properties are defined (subsection 2.3.3). As an example of the classic techniques to take care of long running business transactions a Saga (subsection 2.4.1) and a business transaction model of the business transaction framework are introduced (subsection 2.4.2). Examples how the ACID properties are relaxed are given (subsection 2.5). The expectations for new techniques to take care of long running eBusiness transactions are listed (subsection 2.6).

Many techniques have been developed to take care of the execution of the long running business transactions e.g., Saga and a business transaction framework. They will be introduced as well as their recovery management. The states and the transitions of the long running transaction of the business transaction model will be described. Atomicity types and the phases of the eBusiness transaction will be presented. The implementation of the business transaction framework will be handled briefly. The ways how the Saga and a long running transaction of the business transaction model relax ACID-properties will be described and they will also be compared.

A technique to take care of the recovery of the long running eBusiness transactions can be improved compared to a Saga or a long running transaction of the business transaction model. The things an improved recovery model should take into account will be listed as well as the expectations that some aspects of the execution requirements of an eBusiness transaction set to the service composition model and its recovery model including the demands of relaxed ACID properties. The expectations will be grouped by an aspect of execution requirements of an eBusiness transaction and in some cases further explanation will be provided.

2.1 eBusiness Collaboration and an Example of an eBusiness Process

eBusiness collaboration carries the features automated, cross-organizational, complex and long-running. eBusiness collaboration is expressed using an eBusiness process which consists of services. An example of an eBusiness process is presented below.

Organizations want to focus on their core business and that is why they buy some services from their partners. By combining bought services to its core business processes the organization will reach its business goals [GLA02]. E.g., selling mobile phones is a core business for a telecommunication operator but delivering them is not. So the telecommunication operator will buy a delivery service from a logistics provider. Electronic business, eBusiness methods enable an organization to link its internal data processing systems to the partners' external data processing systems. In order to gain efficiency selecting a partner and linking to its data processing system should be done automatically.

Partner services must fit into an organization's core business process so that it will be one *automated* process [PAP03]. E.g., a selling and delivering process of a mobile phone is depicted in Figure 1A [GLA02]. A telecommunication operator sells a mobile phone to a customer, handles the sale e.g., connects the mobile phone to the network and finalizes the sale after getting a confirmation of the delivery to the customer. Logistics providers offer a mobile phone delivery service to the operator. So selling and delivering a mobile phone to the customer is an automated process for telecommunication operator.

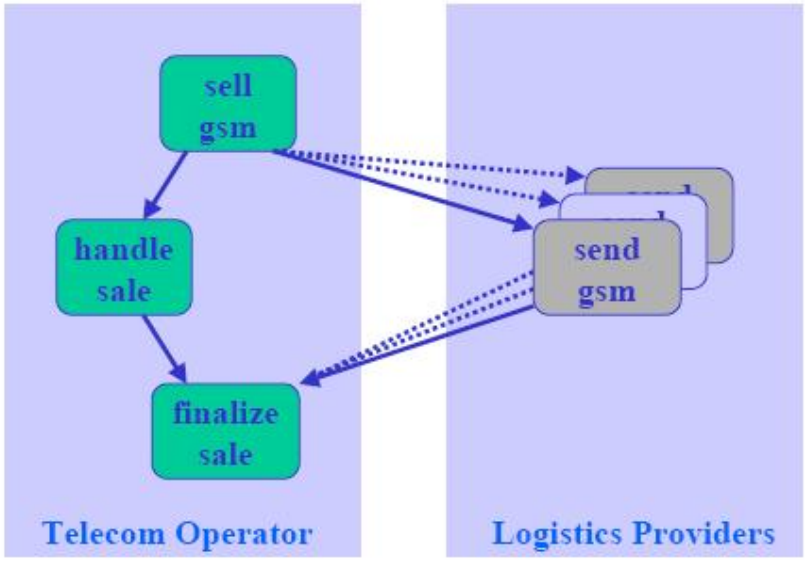

Figure 1A. A selling and delivering process of a mobile phone [GLA02].

To better satisfy the needs and the expectations of its customers, the organization wants to decide as late as possible whose service is the best thinking of e.g., effectiveness, quality and price of the service. The organization determines dynamically the service it utilizes. So there are *several organizations* which offer the same needed service and a service buyer can collaborate with all of them. In Figure 1A, the operator selects and buys a suitable service based on its shipping requirements and current offers from the logistics providers [GLA02]. So the operator collaborates with a selected provider, which can vary each time the delivery service is needed. In order to dynamically choose the best service the provided logistics services must be documented and available whenever requested by a telecommunication operator. That is why the logistics providers publish their services as Web services. E.g., a delivery of a mobile phone can be a Web service which belongs to the selling and delivering process of the telecommunication operator.

An *eBusiness process* of an organization consists of several services and some of them may be implemented as Web services. A *service* is meaningful business functionality and can be bought from different business partner organizations [PAP03]. The service can consist of several Web services [PAP03]. E.g. in Figure 1B above a white line is described a “send GSM” –service of logistics provider (Figure 1A). The “send GSM” -service consists of three services: planning transport, collecting GSM and delivering GSM [GLA02]. Some of the services can be bought from the third business partner organization. The logistics provider’s services: collecting GSM and actual delivery of it can be bought from a post office, UPS, DHL or FedEx etc. A service can be divided into several sub services which can be further divided into several services and additionally there can be many organizations enacting with this wide net of services. This can result to a very *complex* net of services [PAP03]. A service is executed by performing its functions and it terminates even if the other services of the process may or may not terminate successfully. It may take several days to finish a Web service of an eBusiness process. E.g., in Figure 1A the service “send GSM”, which sends a mobile phone to an end customer, can take one to three days depending on how the mobile phone is delivered: by mail or using a courier service. So the service of the eBusiness process is *long-running* if compared to saving one piece of data into a data base.

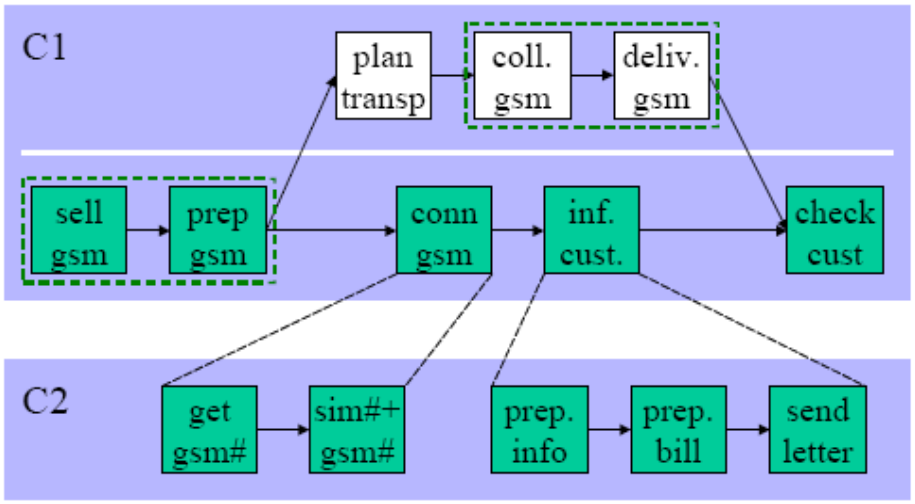

Figure 1B. A selling and delivering process of a mobile phone specified in an abstract technique at the conceptual level [GLA02].

2.2 Management of a Database Transaction Compared to Management of an eBusiness Process

An eBusiness process comprises features of a database transaction. If an operation of the database transaction fails, the transaction needs to be recovered to a consistent state. In a similar way, if a service of the eBusiness process fails while running the eBusiness process, the eBusiness process needs to be restored to a consistent state by performing a backward recovery or a forward recovery. The examples of those two recovery types will be given and their relationship to compensation and to contingency will be introduced. A term eBusiness transaction will be defined in this subsection.

An eBusiness process will be run by activating its services. If an error occurs while executing a service, the service fails. If one of the services fails, the whole process fails. In that case the process needs to either be restored to a consistent state which means either to do a backward recovery [PAP03] or the critical values must be corrected and then continue running which is called a forward recovery [UX09]. Those two options can be compared to the database operations: the backward recovery is a rollback operation and the forward recovery is like manually correcting the wrong data values and after continuing to run the operations.

Examples of the backward recovery and the forward recovery are given here as follows. In the Figure 1B, above a white line is described a send GSM –service of logistics provider:

planning transport, collecting GSM and delivering GSM [GLA02]. If the planning transport –service notices that a number of the stock of the mobile phone to be collected is zero then there are two options. The first option is to cancel the services planning transport and all other services run before it. So the services sell GSM and prepare GSM should be cancelled as well. That is an example of the backward recovery. The second option is to make a phone call to an end customer to ask if she or he can wait so long that there is an ordered mobile phone in the stock. If the customer answers positively the delivery date will be changed to a date when the mobile phone is expected to arrive to stock. When the stock quantity of that mobile phone is one or more, the planning transport service can be finished and the service collecting GSM can be run. That is an example of the forward recovery.

The first option is also an example of the *compensation* in a case when a cancelling of a service is done by running a procedure which will “logically undo” the influence of the partially and/or fully executed functions of the service [UX09] [CZM10]. So the compensation is a form of a backward recovery. The second option can be seen as an example of the contingency because the planning transport –service is handled in an alternative way rather than not just making a plan. Instead of it the customer’s opinion will be asked for and in case of a positive answer the delivery date will be changed. Therefore a *contingency* provides the process an alternative way for continuing to run [UX09] [CZM10] and is a form of a forward recovery [UX09].

Because many Web services use databases, it looks like it is possible to manage eBusiness processes using a classical database transaction concept [SH05]. There are some similarities between eBusiness processes and database transactions, e.g., both manipulate data, consist of operations, maintain continuous records of their activations of operations and have dependencies to other ones. But it has been noticed that for the nature of eBusiness, e.g., long-running and complexity, which were described using the example of the selling and delivering process of a mobile phone the classical database transaction concept is not suitable for eBusiness processes as such [HA02, SH05,UX09].

An idea of a transaction is that it is a model of a set of operations that are executed in a certain order and it is not possible to separate an operation from it because it is one logical operation [SH05,PAP03]. The influence of the transaction will become visible to other transactions when all the operations of that logical operation have been executed

successfully. In case one of the operations of that logical operation fails the situation is comparable to one where nothing has been executed. The idea of the transaction has been taken and the fact that these operations are run by several business partners have been added to it. An *eBusiness transaction* is an interaction between several business partners that are aimed towards achieving a predefined goal [HA02] which is the target of one logical operation [PAP03].

2.3 ACID-properties, Features and Execution Requirements of eBusiness Transactions

A classical database transaction conforms to ACID-properties which are rehearsed (subsection 2.3.1). The features that differentiate an eBusiness transaction from a classical database transaction are introduced as well the additional requirements they set to an execution of an eBusiness transaction compared to the classical database transaction (subsection 2.3.2). The relaxed ACID properties are defined (subsection 2.3.3).

2.3.1 ACID-properties

When the classical data base transaction is executed it obeys ACID-properties: atomicity, consistency, isolation and durability. The ACID-properties are listed in Table 1 [SH05] and further discussed below.

Property	Meaning	Example
Atomicity	All or nothing	If we transfer money within a bank from a source account to a target account then either the money moves from source target or stays where it is.
Consistency	Integrity preserving	If the deposit and withdrawal programs are individually correct then so are all the concurrent executions of them.
Isolation	Hidden partial results	No one can see a state of the database where the money has been withdrawn but no yet deposited.

Durability	Permanent committed results	Once the money has been transferred the state of the accounts is exposed to all.

Table 1. The ACID-properties for traditional transactions [SH05].

Atomicity guarantees that all or none of the operations of the transaction will be run. *Consistency* takes care of that all the operations of the transaction have been executed without affecting harmfully each other. *Isolation* defines when data, which the operations of the transaction handle, will become visible for other concurrent operations. When maintaining the highest level of isolation, usually data is locked until the transaction has been committed or a multiversion concurrency control is implemented. *Durability* guarantees that the results of the transaction are permanent.

2.3.2 Features and Execution Requirements of eBusiness Transactions

An eBusiness transaction has many features which a classical database transaction does not have, e.g., complex, loosely coupled and long-running just to name a couple of them. They set additional requirements for the execution of the eBusiness transaction. The following paragraphs will explain what those features are and what kind of additional requirements they set to an execution of an eBusiness transaction compared to the classical database transaction.

An execution of an eBusiness transaction is performed in several steps and distributed in many systems that make the eBusiness transaction *complex* [HA02] [PAP03]. It causes that conversation between systems must been controlled, a capacity of recourses must been planned and data messages between systems have to be compatible [HA02]. The eBusiness transaction is *loosely coupled* [PAP03] which means that there is a limited number of data and control dependencies between transactions [HA02]. This makes a requirement that a service buyer must be able to use a service without knowing its implementation, internal structure or implementation environment [HA02] [PAP03].The eBusiness transaction is *long-running,* because it often contains long-running services [HA02] [PAP03]. The finishing time of the transaction is difficult to predict. Therefore, a protocol used in execution of the eBusiness transaction must pay attention to duration of execution and business deadlines [HA02].

It is not possible to use a simple rollback to reverse the eBusiness transaction. So it is possible to say that the eBusiness transaction is *difficult to reverse* [HA02]. A protocol used in execution of the eBusiness transaction must have a mechanism for compensation and a contingency plan [HA02]. A result of an eBusiness step, sub-transaction, should be saved on the change that another sub-transaction or even the whole transaction fails, because the result may be valuable in a sense of a complexity, duration or reusability of computation of the result [HA02]. The eBusiness transaction is *recoverable* if logging, save points and context security mechanisms are available [HA02]. The eBusiness transaction is *reliable* if an execution time of each sub-transaction is kept in the promised timeframe [HA02]. So a mechanism for it and for a compensation of each sub-transaction are needed [HA02].

Because there can be many *concurrent* implementations of the same service and also it can be a sub-transaction of another service, the concurrency of the eBusiness transaction is much higher than the concurrency of the traditional data base transaction [HA02]. Therefore, isolation has to be extended [PAP03] to *selected isolation,* which allows that some data elements are not locked during the execution of the eBusiness transaction [HA02]. The selected isolation is later referred to with the name relaxed isolation. Many partner organizations may participate an eBusiness transaction by offering and executing a sub-transaction, e.g., Web service [PAP03] using their own systems. It causes that it is not possible to synchronize these systems [HA02]. Although some protocols require confirmation of every message, the eBusiness transaction is assumed to be *asynchronous* [HA02]. The eBusiness transaction is *reusable* when the requirements of loosely coupling are fulfilled so that each sub-transaction of the eBusiness transaction is encapsulated as a service [HA02]. The reusability of the eBusiness transaction can be improved by defining the cohesive, reusable and transactional constructs [HA02] using unaccustomed classifying of atomicity [CO08]. The unaccustomed classifying of atomicity is described in the atomicity types of the business transaction model which are introduced in Subsection 2.4.2. .

The additional requirements of the execution of an eBusiness transaction introduced above can be put into categories, which describe an aspect of a group of additional requirements [HA02]. The features of an eBusiness transaction and the matched aspects have been listed in Table 2.

A feature of an eBusiness transaction	An aspect of additional execution requirements of an eBusiness transaction
complex	granularity, cohesion
loosely coupled	coupling
long-running	duration, longevity
difficult to reverse	reversibility
recoverable	recoverability
reliable	reliability
concurrent	concurrency
asynchronous	synchronization
reusable	reusability

Table 2. The features of an eBusiness transaction and the matched aspects of the additional execution requirements of it.

2.3.3 Relaxed ACID Properties

The features of an eBusiness transaction demand that the ACID properties must be relaxed for the eBusiness transaction [PAP03][SH05][UX09][CZM10]. The definitions of the relaxed ACID properties are given, to be able to describe how they are relaxed in a Saga and a long running transaction of the business transaction framework when introducing the Saga and the long running transaction of the business transaction framework as classics. *Relaxed atomicity* means that an eBusiness transaction must produce one of the agreeable results defined for that specific eBusiness transaction [UX09] [CZM10]. *Relaxed isolation* demands that the eBusiness transaction does not lock the resources it needs for the whole execution time of the transaction [PAP03] [CZM10]. *Relaxed consistency* means that the state of the eBusiness transaction must satisfy a predefined rule concerning business logic [UX09][CZM10].

2.4 Classic Techniques to Take Care of Long Running Business Transactions

Many techniques have been developed to take care of the execution of the long running business transactions, e.g., Saga [GS87] and a business transaction framework [PAP03]. The concept of the Saga (subsection 2.4.1) and its form to manage concurrently running transactions called a parallel Saga (subsection 2.4.1.2) are introduced. The recovery management of the Saga (subsection 2.4.1.1) and the parallel Saga (subsection 2.4.1.2) are described as well. The business transaction framework and its component a business transaction model are introduced (subsection 2.4.2). A recovery management of the long running transaction of the business transaction model is discussed (subsection 2.4.2.1). The states and the transitions of the long running transaction of the business transaction model are also described (subsection 2.4.2.1). Atomicity types and the phases of the eBusiness transaction are presented (subsection 2.4.2.2). The implementation of the business transaction framework is handled briefly (subsection 2.4.2.3).

2.4.1 Saga – A Technique to Take Care of the Long Running Business Transactions

A *Saga* is a long lived (lasts hours or days) transaction which is composed of sequential transactions that can interleave with other transactions [GS87]. The Saga has only two nesting levels: a Saga which consists of simple transactions and a *simple transaction* consist of actions [GS87]. The simple transaction (later a transaction) is an atomic unit. The actions and the data handling of a transaction and the Saga will be recorded to a log [GS87].

The Saga does not allow partial execution [GS87]. It means that all the transactions of a Saga must be performed successfully or the compensation transactions of the Saga will be run to cancel all the performed transactions of the Saga. So the Saga relaxes the atomicity by defining simple transactions which can be committed or compensated but if a transaction of the Saga fails then the entire Saga must be compensated. A *compensation transaction* C which undoes the semantic meaning of a transaction will be defined for each saga transaction T, but the state of data which the transaction T started to handle may not be returned after running the compensation transaction [GS87]. After the Saga T_1, T_2, … ,T_n has got the compensation transactions C_1, C_2,…,C_{n-1} either a series T_1, T_2, … ,T_n or a series T_1, T_2, … , T_j, C_j, ..., C_2, C_1 for some $0 <= j < n$ will be performed [GS87]. In the latter case it is possible that other transactions will see the result of T_2, but while performing the

compensation transaction C_2 there will not be any attempt to inform or abort them [GS87]. The Saga may show also its partial results for other Sagas [GS87]. So the Saga relaxes isolation by letting a transaction commit without taking into account has the other transactions of the same Saga committed [SHH05] and letting other Sagas to use the result of that committed transaction. The Saga also relaxes the consistency by defining a compensation transaction for each transaction of a Saga, but the Saga does not solve the inconsistency problem which is resulted from the relaxation of the isolation of the Saga [SHH05].

A user can start an *abort-saga* command which will terminate the current transaction and the whole Saga by performing the compensation transactions or the user can abort only the current transaction with *abort-transaction* command [GS87]. Between executions of the transactions of the Saga the user can save the state of data by giving a save-point command and that state , *save-point*, can be used as a starting point of the compensation transactions in case of the need to restart the Saga [GS87].

2.4.1.1 Recovery Management of a Saga

If a failure happens in the execution of a Saga there are three choices for recovery. The first choice is to do a backward recovery by running the compensation transactions. Nevertheless, it may cause an inconsistency problem, because other Sagas may have used the result of a committed transaction of the failed Saga before it will be compensated. The second choice is to do a *pure forward recovery* by starting performing the missing transactions from the most recent save-point of the Saga if the save-point command is run automatically at the beginning of each transaction [GS87]. So the pure forward recovery does not use the compensating transactions and that is why it is a better recovery option than the first option. There might be situations that the forward recovery is not possible [GS87]. The third choice is to do so-called *backward-forward recovery* [GS87] where the after the save-point successfully performed transactions must first be compensated and then start running the transactions again from the save-point. Eg., a Saga is composed of four transactions T_1, T_2, … ,T_4 and it has the compensation transactions C_1, C_2, C_3. The Saga is executed by running T_1, T_2, giving a save-point command and running T_3 and T_4 [GS87]. While running the transaction T_4 there occurs a failure. A recovery will be done by aborting T_4 and running C_3 that is a backward recovery to the save-point and then running T_3 and T_4 again that is a forward recovery. The backward-forward recovery is also better solution than backward recovery because it starts from a save-point which is a consistent

state of data. If the save-point command is run automatically at the beginning of each transaction and abort-saga command is not allowed then backward recovery is never needed [GS87]. It is useful in the situations that it is difficult to define a compensation transaction [GS87].

If there is severe malfunction in the system, the backward recovery is done as follows [GS87]. The pending transactions are either aborted or committed. If all the transactions of a Saga are committed then the Saga is committed. Otherwise the Saga will be aborted: the last successfully performed transaction will be compensated as well as all the preceeding ones [GS87].

If a failure happens in the execution of a compensation transaction or in the pure forward recovery there are three options. The first option is to reset the system to the state it was before the execution of a compensation transaction or the pure forward recovery were started and after that *retry* the failed transaction [GS87]. The second option is to run an *alternate transaction* which will produce the same result than the original transaction however using a different algorithm or a technique. The third option is a *manual intervention,* which means that the failed transaction is aborted, the implementation of the transaction is changed based on the description of the failure and the transaction is rerun [GS87]. During the manual intervention the Saga is pending until the rerun has been started [GS87]. It must be noticed that the Saga will not hold any data resources while the implementation of the transaction is changed [GS87]. This is not good because while repairing the implementation of the failed transaction other Sagas may have affected to the same data resources. When the rerun of the failed transaction will be started the transaction may return different result than it would have returned without the failure. So this third option of the forward recovery of the Saga is not very successful way to relax the isolation, because it may cause a consistency problem.

2.4.1.2 A Parallel Saga

A *parallel Saga* is a Saga having transactions which can be performed concurrently [GS87]. While running a parallel Saga it (the parent Saga) will create a new Saga, a child Saga [GS87]. If there is a crash in the system the backward recovery is done quite similar way than in the sequential Saga but the compensations of the child Saga are run before any of the transactions of the parent Saga which were performed before the child Saga was created [GS87]. Otherwise, the running order of the compensations follow only the

original performance order of transactions and create child Saga actions within one Saga [GS87]. E.g., there is a parallel Saga T_1,T_2, T_3 and a child Saga of it T_a, T_b has been created after T_1 has been performed. T_a has been run after T_2 and red the data written by T_2. If they must be compensated T_a compensation is done regardless of when the compensation of T_2 is done.

A backward recovery of a parallel Saga is more challenging than a backward recovery of a Saga with sequential transactions, because there can be many child Sagas which must be handled as well [GS87]. The backward recovery of a parallel Saga starts so that a child Saga sends an abort-saga command. Then all other child Sagas and the parent Saga will be terminated, the pending transactions will be aborted and all committed transactions will be compensated [GS87].

A forward recovery of a parallel Saga is more difficult than the backward recovery, because the save-points of the child Sagas and parent Saga may not be analogous [GS87] and that way not suitable for to restart each child Saga from the save-point. Figure 2A [GS87] depicts that kind of situation. A parent Saga is composed of the transactions T_0, T_1, T_2, T_3 and a save-point command have been run before the transaction T_1. A child Saga has been created after the successful execution of T_1. The child Saga is composed of the transactions T_4, T_5 and a save-point command have been run before the transaction T_5. A failure occurs during the execution of the transactions T_3 and T_5. A forward recovery of the parent Saga can be started from its save-point but there is no use for the save-point of the child Saga. This problem is called *cascading rollback* [HV82]. The parallel Saga solves the problem by keeping a log of the order of execution of the transactions, creation of child Sagas and run save-point commands [GS87]. The log is used to find the latest save-points of the child Sagas and the parent Saga which no earlier transaction of another Saga has to be compensated after it [GS87]. Such a save-point is the save-point of the parent Saga in the Figure 2A. If that kind of save point does not exist all the transactions of the parent Saga and all the child Sagas must be compensated [GS87]. If such save-points are found then the backward recoveries are done up to them and after that the rerunning of them is started [GS87].

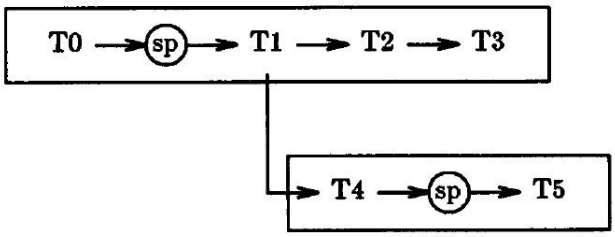

Figure 2A. A parallel Saga. [GS87]

2.4.2 Business Transaction Framework and a Business Transaction Model

A consistency and reliability of a business process which consist of web services will be raised by defining long running business transactions between web services [PAP03]. A web service consists of several operations which have transactional properties [PAP03] and should be handled as a logical part of the long running business transaction. The transactional properties of a web service affects to the transactional behavior of the entire business process [PAP03]. A *business transaction framework* (BTF) has been developed to support the transactional behavior of a business process [PAP03]. The usage of the BTF makes possible to orchestrate the loosely coupled Web services so that they are executed in a single business transaction [PAP03]. That supports the coordination of the Web services of a business process and ensures the co-operation of the business partners of the process.

The BTF consist of three components: a business transaction model, coordination protocols and business protocols [PAP03]. The *business transaction model* (BTM) describes long running eBusiness transactions, ordinary transactions, exception handling mechanisms, compensating actions and atomicity criteria for business [PAP03]. The *coordination protocols* coordinate the operations of Web services across distributes systems using transactional mechanisms [PAP03]. A system should also be able to spread out an operation to other services and to register for coordination protocols [PAP03]. The heterogeneity of protocols of the partners' own workflow and transaction management systems to coordinate the operations and interoperation of business transactions are hidden [PAP03]. The *business protocols* qualify content, a purpose and an order of the business messages to be sent between partners concerning actual eBusiness [PAP03].

2.4.2.1 A Business Transaction Model (BTM)

There are two kinds of transactions in the BTM: atomic and long running [PAP03]. An *atomic transaction* is that way similar to a Saga that they are both atomic: either the effect of the transaction is committed or rolled back. If a failure occurs during the execution of the atomic transaction the operations of the atomic transaction are compensated as in the first recovery option of the Saga. The backward recovery of the atomic transaction is done automatically [PAP03]. The atomic transaction differs from Saga so that it can have a nested structure using a close nesting model [PAP03] (all transactions are performed or none) [CO08] but the Saga can have only two nesting levels. An atomic transaction uses the two phase commit protocol [PAP03].

A *long running transaction* is composed of the atomic transactions and allows a partial execution which is not possible in Saga. The *partial execution* is done using an open nested transaction model [CO08] which means that some of the atomic transactions of the long running transaction are committed and some although they could commit are cancelled by doing a roll back [PAP03]. The decision to commit or to roll back an atomic transaction is made individually by participants of the long running transaction [PAP03] [CO08]. The participants do not get a similar result of the atomic transaction. So the nested transaction model of long running transaction relaxes the atomicity.

If a long running transaction fails, it runs a backward recovery by starting some compensating activities [PAP03] which will reverse the effects of the failed transaction. The backward recovery returns the entire process to the consistent state including the possible child transactions of the failed transaction [PAP03]. The participants do not know the eventual outcome of the long running transaction when they confirm or cancel an atomic transaction of it which means that the results of the atomic transactions are not isolated [CO08]. So the partial results of the long running transaction relax isolation.

Furthermore, the atomic transactions may have used the partial results of each other or other long running transactions may have used them [CO08] and a recovery is needed to return a long running transaction to a consistent state [PAP03] [CO08]. A backward recovery by compensating the atomic transactions is not a suitable option, because the atomic transactions of long running transaction may have caused the side effects which are not reversible [PAP03]. That is why business logic of the long running transaction must be qualified to execute the backward recovery of the long running transaction

[PAP03]. If a system failure occurs during the execution of the long running transaction the atomic transactions are guaranteed a successful forward recovery and the long running transaction performs a forward recovery by returning a consistent state and continuing catering for the occurred failure [PAP03]. So the backward recovery together with compensation and definition of the business logic of the long running transaction and also the forward recovery are successful ways to relax consistency. Both of them guarantee a consistent state for the long running transaction [PAP03].

Each instance of the long running transaction of the BTM has its own *transaction context* [PAP03]. It describes a nesting structure (parent – child relationships), a type (atomic or long running) and all executed activities (including child contexts) of the instance [PAP03]. The child contexts are included for the possible compensation of the transaction after the transaction has completed [PAP03]. The instance can be in different states, which are changed by doing a transition. The states and the transitions of the instance are illustrated in the Figure 2B [PAP03] and described below.

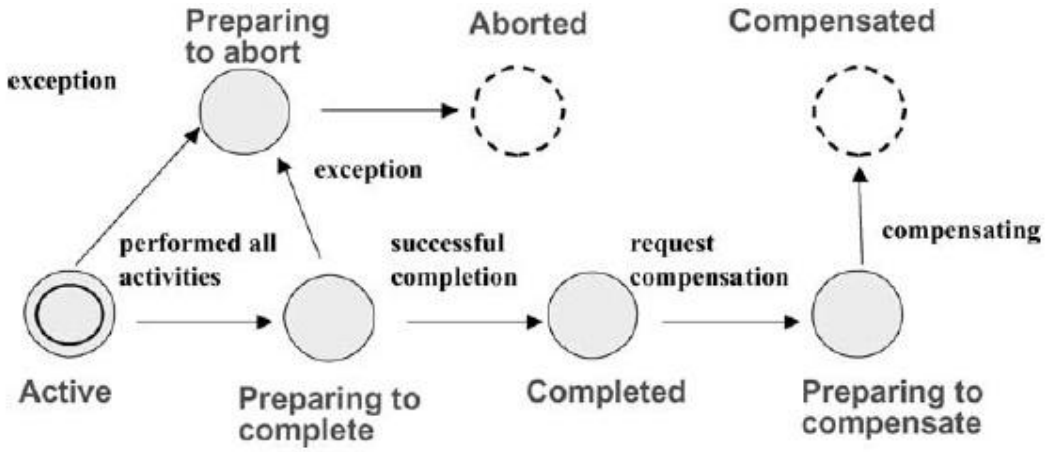

Figure 2B. The states and the transitions of a long running transaction of the business transaction model of the BTM [PAP03].

The first state of the instance is *active*. Then it performs activities of its transaction context [PAP03]. After the instance have been performed all its activities it becomes to the state *preparing to complete*. In that state the data changes may be marked persistent, a two-phase commit executed for an atomic transaction and the nesting structure coordinated for a long running transaction [PAP03]. After everything is done to be able to complete the instance successfully the instance moves to the *completed* state. If an exception has occurred while performing activities the instance will move from the active state to a

preparing to abort. In that state the compensation transactions may be run, the result of the atomic transaction told to the participants and an abortion of the nested long running transaction coordinated [PAP03]. After everything is done to abort the instance the instance moves to the *aborted* state. If the compensation of the instance is asked the instance will move from the completed state to the state *preparing to compensate*. In that state the compensation activities are executed. When all of them are run successfully the instance becomes *compensated*. The final states of the instance are aborted and compensated. Even though Figure 2B does not show it, completed can be a final state of the instance because if the compensation is not required the instance will stay in the state completed.

2.4.2.2 Atomicity Types and the Phases of the eBusiness Transaction

Atomicity types [PAP03] [CO08] also called unaccustomed classifying of atomicity [HA02] can be used to define the cohesive, reusable and transactional constructs which improve the reusability of the eBusiness transaction [HA02]. The idea is to guide the eBusiness transaction taking into account the systems which use it [PAP03]. There is a *service request atomicity* which guarantees for a service buyer that the eBusiness transaction is an atomic work flow [HA02] [PAP03]. The *conversation atomicity* defines who structures, monitors and controls the conversation between a service buyer and a provider [HA02] [CO08]. The *non-repudiation atomicity* specifies the non-repudiation provisions [HA02] using digitally signing in the content of a transaction [CO08] which helps a service provider to handle disputes [PAP03]. The *contract atomicity* takes care of that a service buyer and a provider have made a contract [HA02] [CO08] and agreed legal terms and conditions and technical specifications of their collaboration [PAP03]. If a transaction is a contract atomic one then it is automatically a conversation atomic [PAP03]. There is no context atomicity type in the BTM. The *context atomicity* confirms that there is an implemented mechanism to ensure correctness of context of the eBusiness transaction during the execution [HA02].

The *payment atomicity* indicates for a service buyer to pay for the service [HA02] and has an influence to money transfer [PAP03] [CO08] . The goods delivery atomicity demands that goods or services are identifiable [HA02] and can be delivered as agreed beforehand [PAP03] [CO08]. The *certified delivery atomicity* guarantees a delivery of correct goods [PAP03] [CO08]. It is possible to reach if a document of the delivery can be send to the all involved parties of eBusiness transaction [HA02]. The introduced atomicity types can be

used to convey the semantics of the business system and can be defined using XML constructs [PAP03]. The atomicity types make it possible to use abstract terms to reach the compliance goals and to make them concrete [CO08]. For that a framework which maps the atomicity types of a transaction onto basic transactions has been brought forward [PK06].

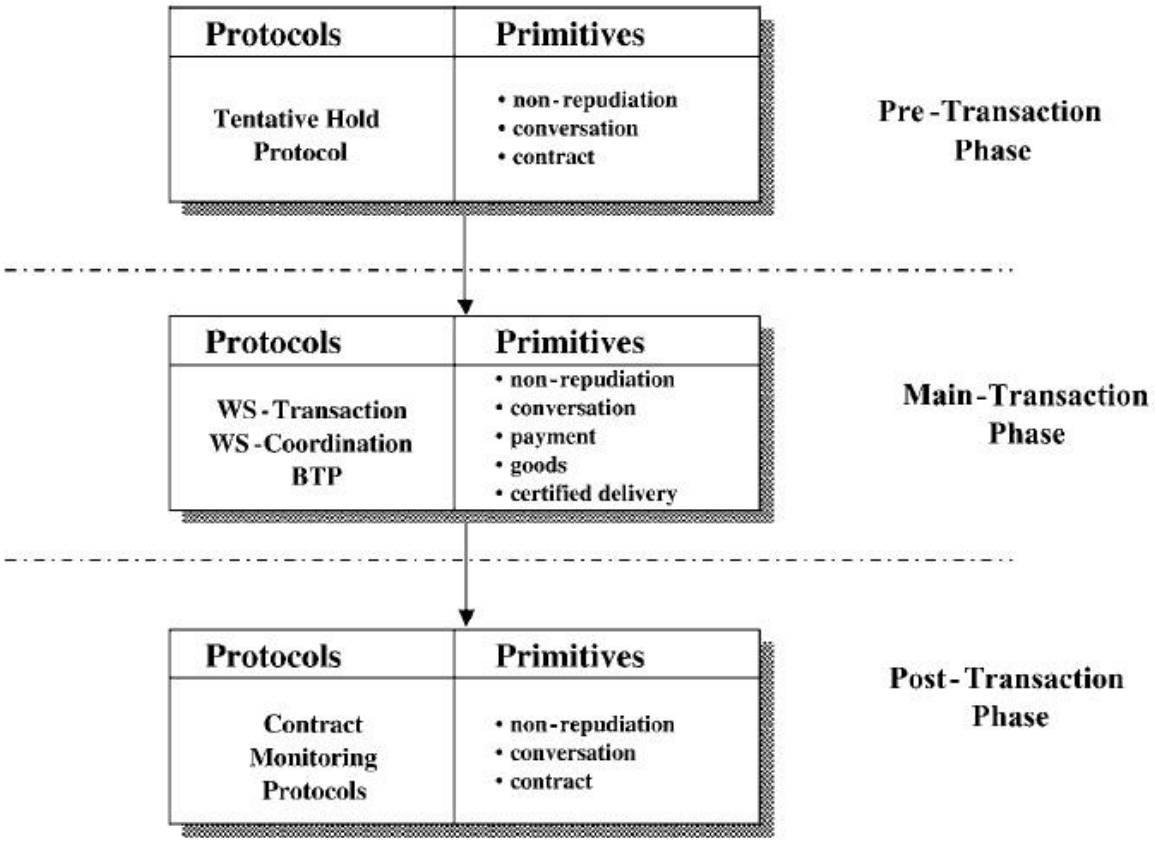

Figure 2C. The phases of an eBusiness transaction and the atomicity types used during them. [PAP03]

All atomicity types are not relevant in all transaction phases [HA02] [PAP03]. That's why an eBusiness transaction is organized in three phases: pre-transaction, main-transaction and post-transaction phase [PAP03][HA02]. The *pre-transaction phase* is the time before the execution of the eBusiness transaction, *post-transaction phase* is the time after the execution and the *main-transaction phase* is between them [HA02] [PAP03]. The phases of an eBusiness transaction and the atomicity types used during each phase are depicted in the Figure 2C [PAP03]. During the pre-transaction phase the information needed to execute the business transaction is changed between the trading participants e.g. order

information, prices and delivery conditions [PAP03]. The operations run during the pre-transaction phase can implement non-repudiation, conversation and contract atomicity. A business process and its transactions are mainly run during the main-transaction phase. They use those parts of the BTF which offers the protocols and infrastructure which coordinates the execution of Web services [PAP03]. The operations run during the main-transaction phase can implement non-repudiation, conversation, payment, goods and certified delivery atomicity. The fulfillment of the contracts and terms and conditions defined during the execution of the eBusiness transaction will be monitored in the post-transaction phase [PAP03]. The operations run during the post-transaction phase can implement non-repudiation, conversation and contract atomicity.

2.4.2.3 Implementing Business Transaction Framework (BTF)

The BTF needs a Web service orchestration infrastructure which allows dynamic service compositions of Web services in a business process, data flow coordination and the usage of the exception and error handling mechanisms [PAP03]. Those functionalities belong to the BTM of BTF. The Web service orchestration infrastructure must also support the business transaction and coordination mechanisms and business protocols [PAP03] which are described in the coordination protocols component and the business protocols component of the BTF.

The three components of BTF can be implemented on the Web service orchestration infrastructure which can be described using the standard Business Process Execution Language for Web Services (BPEL4WS) [PAP03]. *BPEL4WS* is a language to define business processes and business interaction protocols [CUR03]. BPEL4WS offers all the functions that the BTM needs [PAP03]. BPEL4WS have been developed further and nowadays it is known as Web Services Business Process Execution Language (WS-BPEL) [AL07].

The group of three standards can be used to implement the coordination protocols component and the business protocols component of the BTF. The group of three standards describes mechanisms for Web services domains to take care of their transactional interoperability and offers for a service the way to comprise transactional qualities into Web services applications [CAC05]. *WS-Coordination* standard represents an extensible coordination framework [CAC05] which uses two different coordination types: WS-AtomicTransaction for short duration, ACID transactions and WS-

BusinessActivity for long running business transactions. *WS-AtomicTransaction* standard defines three specific agreement coordination protocols: completion, volatile two-phase commit and durable two-phase commit [CAA05]. Any or all of these protocols can be used in applications that need to have consistent agreement of the result of the short duration distributed operations having atomicity property [CAA05]. *WS-BusinessActivity* standard defines two specific agreement coordination protocols: BusinessAgreementWithParticipantCompletion and BusinessAgreementWithCoordinatorCompletion [CAB05]. One or both of these protocols can be used in applications that need to have consistent agreement of the result of the long running distributed operations [CAB05].

2.5 Relaxing of the ACID Properties – Examples and Comparisons

An eBusiness transaction has a relaxed form of atomicity, consistency and isolation. To give examples, the ways how they are implemented in Saga and in a long running transaction of BTM are described and they are also compared in this subsection.

Relaxed atomicity means that the eBusiness transaction must produce one of the agreeable results defined for that specific eBusiness transaction, because there might be a situation that an operation of a transaction fails but it is still not necessary to cancel the entire transaction [PAP03]. A Saga relaxes atomicity by defining simple transactions which can be committed or compensated. If the transaction of the Saga fails then, however, all the transactions of the Saga must be compensated. That is why the Saga does not properly relax atomicity. A long running transaction of the BTM implements the relaxed atomicity [CO08] using open nested transactions. It means that the participants of the long running transaction decide individually either to commit or to roll back an atomic transaction of the long running transaction [PAP03].

Relaxed isolation demands that the eBusiness transaction does not lock the resources it needs for the whole execution time of the transaction, because no other business partners can take a part in the transaction [PAP03] if the recourses are locked for only one partner for the whole time of the execution of the transaction. A transaction of a Saga can commit without taking account has the other transactions of the same Saga committed. That is how the Saga relaxes isolation. The isolation can be relaxed by allowing partial results of an eBusiness transaction. The long running transaction of the BTM implements the relaxed isolation [CO08] using a partial execution of long running transaction [PAP03].

Relaxed consistency is reached if the state of an eBusiness transaction satisfies a predefined rule concerning business logic. It can be implemented if the transaction can be returned to a consistent state when preparing a forward recovery after a failure a long running eBusiness transaction [PAP03]. The Saga can do the implementation if the save-point command is run automatically at the beginning of each transaction. The consistent state can also be returned in the backward situation by using compensation and defining business logic of the long running transaction [PAP03]. The Saga relaxes consistency by defining a compensation transaction for each transaction, but the Saga does not solve the inconsistency problem caused by simultaneous relaxation of isolation. So the consistency can be relaxed using a forward recovery or the backward recovery together with compensation and definition of the business logic of the long running transaction. Those two ways are implemented in the long running transaction of the BTM. The ways how a Saga and a long running transaction of the BTM relax atomicity, isolation and consistency are collected in Table 2B and they will be compared below.

Although each transaction of the Saga is atomic, the Saga allows a transaction to access to the shared recourses [CB97] which means that the Saga relaxes atomicity. But the Saga does not relax an atomicity very well, because if a transaction of the Saga fails then all the transactions of the Saga must be compensated. The long running transaction of the Business transaction model of BTF relaxes atomicity better, because the participants of the long running transaction can individually decide either to commit or to roll back an atomic transaction of the long running transaction.

The Saga relaxes isolation by letting a transaction of a Saga commit without taking into account has the other transactions of the same Saga committed. If other Sagas use the result of the committed transaction before another transaction of the same Saga fails and the entire Saga will be compensated it causes an inconsistency problem which the Saga cannot solve. The long running transaction of the Business transaction model of BTF relaxes isolation basically similar way by allowing the participants of a long running transaction to confirm or cancel an atomic transaction of it without knowing the eventual outcome of the long running transaction. But a difference is that a long running transaction can handle the fail of a transaction. It is handled using a forward recovery or the backward recovery together with compensation and definition of the business logic of the long running transaction.

	Saga	Business transaction model
Relaxes atomicity	*Yes but* if a transaction of the Saga fails then all the transactions of the Saga must be compensated.	*Yes* by an open nested transaction of a long running transaction.
Relaxes isolation	*Yes* by letting a transaction commit without taking into account has the other transactions of the same Saga committed and letting other Sagas to use the result of that committed transaction. That may cause an inconsistency problem which the Saga cannot solve!	*Yes* by allowing the participants of a long running transaction to confirm or cancel an atomic transaction of it without knowing the eventual outcome of the long running transaction.
Relaxes consistency	*Alternatively: Yes* by defining a compensation transaction for each transaction of a Saga, but the Saga does not solve the inconsistency problem caused by simultaneous relaxation of isolation. *or Yes* by using a pure forward recovery if the save-point command is run automatically at the beginning of each transaction.	*Yes* by using a forward recovery which returns a long running transaction to a consistent state and continues the performance taking into account the occurred failure or by doing the backward recovery together with compensation and definition of the business logic of the long running transaction.

Table 2B. A comparison of the ways to relax atomicity, isolation and consistency between a Saga and a long running transaction of the BTM.

The Saga relaxes consistency by returning the Saga to a consistent state when preparing a forward recovery after a failure if the save-point command is run automatically at the beginning of each transaction of the Saga. If that kind of automation is not in use then the Saga tries to reach the consistent state by defining a compensation transaction for each transaction and by running them to do a backward recovery. Because it does not undo the possible side-effects the starting state of Saga will not be fully reached. The long running transaction of the BTM relaxes consistency better, because when doing the backward

recovery using compensation the definition of the business logic of the long running transaction is used too. Then the state of the long running transaction is closer to the starting state of the long running transaction. In a case of forward recovery the long running transaction of the BTM will reach a more consistent state than the Saga because it takes into account the occurred failure when it continues running the forward recovery.

2.6 Expectations for New Techniques to Take Care of Long Running eBusiness Transactions

The things an improved recovery model should take into account are listed as well as the expectations that chosen aspects of the execution requirements of an eBusiness transaction set to the service composition model and its recovery model including the demands of relaxed ACID properties. The expectations are grouped by an aspect of execution requirements of an eBusiness transaction and in some cases further explanation is provided.

A technique to take care of the recovery of the long running eBusiness transactions can be improved compared to a Saga or a long running transaction of BTM. An improved recovery model should take into account the following things. A recovery management has to be built on the top of a service composition model which is robust but flexible. It means that the service composition model has to be hierarchical and well defined starting from a service up to the expression of the entire eBusiness process. Flexible meaning that a transaction model of the process follows open nested transaction. A transaction has to also relax ACID properties. A recovery mechanism should be more detailed described and it has to restore a transaction to a consistent state regardless of when a service fails. A service should have compensation and contingency and the forward recovery should be maximized. The chosen aspects of the execution requirements of an eBusiness transaction set the expectations for the service composition model and its recovery model. These expectations are listed in Table 2C including the demands of relaxed ACID properties and all other things described in this paragraph. The expectations are grouped by an aspect of execution requirements of an eBusiness transaction and further explained below when needed. Even if the Saga and the LRT of the BTM both fulfill an expectation, the expectation is taken to the list as a minimum requirement.

An aspect of execution requirements of an eBusiness transaction	Expectations for a service composition and recovery model
granularity, cohesion	**Granularity** levels from process to a service allowing a **flexible hierarchical** composition structure. A transaction has **relaxed atomicity.**
coupling	Transactions have a limited number of data and control **dependencies** which can be taken into account in the backward recovery.
reversibility	A service has a mechanism for compensation and a **contingency** plan. A forward recovery is **maximized**.
reliability	A mechanism for a compensation of each **sub-transaction** is needed.
concurrency	**Isolation** is **relaxed** which allows that data elements are not locked during the execution of the transaction.
recoverability	A transaction has **relaxed consistency**. There have to be logging, save points and context security mechanisms available so that a transaction reaches a **consistent state** if a service fails. A **recovery mechanism** is described in detailed.
reusability	**Unaccustomed** classifying of **atomicity** is used.

Table 2C. The expectations for the service composition model and its recovery model grouped by the aspects of the execution requirements of an eBusiness transaction.

An expectation for a service composition and recovery model is written first *cursive* and after it in some cases also further explanation is provided. ***Granularity*** *levels from process to a service allowing a* ***flexible hierarchical*** *composition structure.* A recovery model is founded on a service composition model which should be hierarchical and well defined starting from a service up to the expression of the entire eBusiness process. It should

allow a nested composition structure which has a needed number of nesting levels to describe an entire process using services. *A transaction has* ***relaxed atomicity***. This means that the atomicity of the transaction has to be relaxed. *Transactions have a limited number of data and control* ***dependencies*** *which can be taken into account in the backward recovery.* The data and control dependencies are needed in order to take into account the backward recovery of a transaction. *A service has a mechanism for compensation and a* ***contingency*** *plan. A forward recovery is* ***maximized****. A* forward recovery should be maximized and it is explained in Section 5. An explanation is not needed for the next expectations. *A mechanism for a compensation of each* ***sub-transaction*** *is needed.* ***Isolation*** *is* ***relaxed****, which allows that data elements are not locked during the execution of the transaction. A transaction has* ***relaxed consistency****. There have to be logging, save points and context security mechanisms available so that a transaction reaches a* ***consistent state*** *if a service fails. A* ***recovery mechanism*** *is described in detailed. Unaccustomed classifying of atomicity is used.* The new techniques take care of recovery of long running eBusiness transactions and the way how they answer to the above expectations is handled in Section 5. As new techniques the DeltaGrid composition and recovery model is introduced in Section 3 and Constraint condition and logic rule based recovery mechanism is introduced in Section 4.

3 The DeltaGrid Service Composition and Recovery Model

A DeltaGrid environment is introduced by defining the essential terms of it and a Delta-Enabled rollback is defined (subsection 3.1). DeltaGrid service composition model is described and an example of an eBusiness process using atomic and composite groups is given (subsection 3.2). An ACID DEGS operation and a multilevel DEGS operation are defined and their execution is described (subsection 3.3). A pre-commit recoverability mechanism (subsection 3.3.1.1) and a post-commit recoverability mechanism (subsection 3.3.1.2) of a DEGS operation are introduced. An execution of an atomic group (subsection 3.3.2) and a composite group (subsection 3.3.3) are explained as well. A criticality attribute is defined (subsection 3.3.2) and the terms a shallow compensation and a deep compensation are extended (subsection 3.3.3). A backward recovery of the atomic group and the composite group are explained (subsection 3.3.4).

The following algorithms are presented and explained: to make a choice between DE-rollback or service reset for an operation (subsection 3.4), a post-commit recovery for an atomic group (subsection 3.4.1), deep compensation for a composite group(subsection 3.4.2), a recovery of an operation from a failure in the context of a running process (subsection 3.4.3) and also for the atomic group to propagate a failure (subsection 3.4.3). The conditions for the applicability of the different recovery options are defined and the failure recovery algorithm of the operation is demonstrated using an example (subsection 3.4.3).

3.1 The DeltaGrid Environment

A DeltaGrid environment is introduced by defining the essential terms of it, e.g., Delta-Enabled Grid Service, a delta and a delta schedule. An action called Delta-Enabled rollback is introduced and an example of it is given in this subsection.

DeltaGrid environment has a foundational concept called a *Delta-Enabled Grid Service* (DEGS) [UXB09]. It is a Grid Service which has been enlarged with an interface which allows access to the incremental data changes called *deltas* linked to an execution of a service of a process [UX09]. The DEGS produces the deltas and sends them to a *process history capture system* (PHCS) which maintains the execution context of every running

process and creates a time-ordered schedule of data changes [UXB09] for concurrently running processes in the system. A *delta schedule* is a global log file to analyze data dependencies of concurrently running processes [UX09]. Many processes may have interleaved access to the same data recourse, because of relaxed isolation. When a process fails the log file is used to decide how the failure and the recovery of the failed process influence the other processes which have used the same data [UX09].

The delta schedule supports an action called *Delta-Enabled rollback* (DE-rollback) which restores the results of the execution of a service as they were before the execution of the service even if the execution has already been terminated [UX09]. An example of the DE-rollback is depicted in the Figure 2 [UX09]. There are two processes p_1 and p_2 which have two services. The process p_1 has services op_{11} and op_{12} and p_2 has op_{21} and op_{22}. Both processes access data X and Y. The schedule of data changes of X are marked x_1, x_2, x_3 and for Y y_1 and y_2 [UX09]. When the service op_{21} performs DE-rollback the value of X will be changes from x_3 to x_2. Also when the service op_{22} performs DE-rollback the value of Y will be changes from y_2 to y_1. Because the data values are restored in the opposite order that the changes were made DE-rollback can only be run if it fulfills the semantic conditions of the traditional recovery [UX09] later referred as *semantic recovery condition*. The delta schedule takes care of that no dirty writes or reads happen [UX09]. While doing so, op_{22} cannot perform DE-rollback if another service has read the value y_2 of Y. In case it is not possible to use DE-rollback, compensation is needed [UX09].

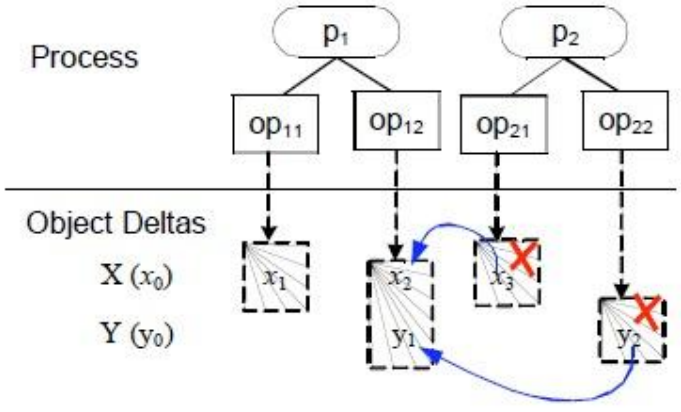

Figure 2. Delta-Enabled rollback (DE-rollback) [UX09].

3.2 The Service Composition and Recovery Model

The execution entities of a service composition model are defined and DeltaGrid service composition structure is described. An example of an eBusiness process using atomic and composite groups is given in this subsection. Also an example is given on how the subscripts of groups show their nesting level within the process.

An eBusiness process consists of different kind of execution entities in the DeltaGrid environment. The *execution entities* describe a service composition model which will express the eBusiness process as a hierarchical structure. There are altogether seven execution entities. The definitions and the denotations of them are described in Table 3 [UX09].

The name of the execution entity	Definition	Denotation
Process	A top level composite group	$\mathbf{p}_i$
Operation	A DEGS service invocation	op_{ij}
Compensation	An operation that is used to undo the effect of a committed operation	$\mathbf{c}op_{ij}$
Con**t**ingency	An operation that is used as an alternative of a failed operation op_{ij}	$\mathbf{t}op_{ij}$
Atomic **g**roup	An execution entity that is composed of a primary operation (op_{ij}), an optional compensation (cop_{ij}) and an optional contingency operation (top_{ij})	$\mathbf{ag}_{ij}$=<op_{ij} [,cop_{ij}] [,top_{ij}]>
Composite **g**roup	An execution entity that is composed of multiple atomic	$\mathbf{cg}_{ik}$= <($ag_{i,k,m}$ | $cg_{i,k,n}$)$^+$ [,cop_{ik}] [,top_{ik}]>

	groups or other composite groups. the composite group can also have an optional compensation and an optional contingency	
DE-rollback	An action to undo the effect of an operation by restoring the data values as they were before the operation	**d**op_{ij}

Table 3. The definitions and denotations of seven execution entities of a service composition model [UX09].

On the highest level of the service composition model there is an execution entity *process* which is a composite group which consists of other execution entities. A denotation of the process is p_i where p is the process and the subscript i is used to identify the process. A single activation of a service of a process is called an *operation*. Its denotation is op_{ij} where op is the operation, the subscript i is used to refer to the surrounding process p_i and the subscript j is used to identify the operation within the process p_i.

A *compensation* is an operation, which will undo the effect of a committed operation. It is presented cop_{ij} where op_{ij} is a committed operation of the process p_i. A *contingency* is an alternative operation which can be run instead of a failed operation. It is expressed as top_{ij} where op_{ij} is the failed operation of the process p_i. So the compensation is a backward recovery and the contingency is a forward recovery in the service composition model. A relationship between a process, an operation, a compensation, a contingency and other execution entities is depicted in an UML diagram of the DeltaGrid service composition structure in the Figure 3 [UX09].

The service composition model makes it possible to define a complex control structure for an eBusiness process by adding scopes in the context of the process execution [UX09]. It is handled by dividing the process into logical execution parts using an atomic group and a composite group. The *atomic group* consists of an operation, an optional compensation and an optional contingency which are shown in Table 3. The *composite group* consists of

many atomic groups or other composite groups which will be run sequentially or in parallel. The composite group also has an optional compensation and an optional contingency. Both the atomic group and the composite group having optional compensation and optional contingency, which makes it easy to recover them from a service execution failure. A denotation of an atomic group is ag_{ij} and for a composite group it is cg_{ik}. The subscripts show the nesting levels within the process p_i. There is a picture of a sample process using atomic and composite groups in an eBusiness process in the Figure 4 [UX09]. It also gives an example how the subscripts of groups show their nesting level within the process.

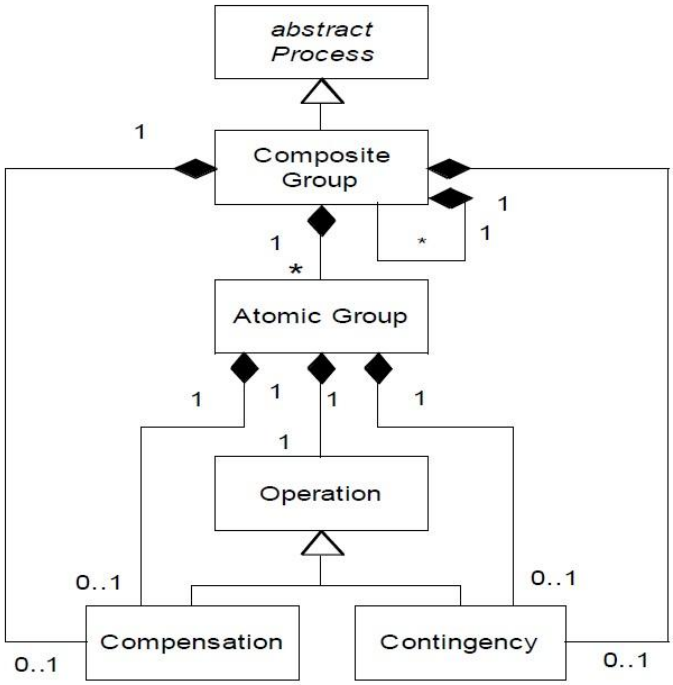

Figure 3. An UML diagram of the DeltaGrid service composition structure [UX09].

A graph of a process p is shown in the Figure 4. As per the definition of a process, a process p is a top level composite group and can then be marked $p_1 = cg_1$. The process p_1 consists of two composite groups $cg_{1,1}$ and $cg_{1,2}$ and an atomic group $ag_{1,3}$. The composite groups $cg_{1,1}$ and $cg_{1,2}$ are composed of atomic groups as follows $cg_{1,1}$: $ag_{1,1,1}$, $ag_{1,1,2}$, $ag_{1,1,3}$ and $cg_{1,2}$: $ag_{1,2,1}$, $ag_{1,2,2}$. So the third subscript of $ag_{1,1,2}$ shows that the atomic group $ag_{1,1,2}$ is on the third nesting level and it is the second group on that level. The atomic group $ag_{1,1,2}$ has a operation $op_{1,2}$ and an optional compensation $cop_{1,2}$ but it

does not have a contingency. The atomic group $ag_{1,1,1}$ and the composite group $cg_{1,1}$ both have an optional compensation and an optional contingency. Please note that in case of the composite group the optional compensation has been marked $cg_{1,1}$.cop instead of $cop_{1,1}$ the denotation of a compensation defined in Table 3.

DE-rollback is an execution entity. It is a system-initiated action to undo the effect of an operation by restoring the data values as they were before the operation. To reverse the execution of the operation, DE-rollback uses the deltas of the PHCS. It is the only execution entity that does not exist in Figure 3. The idea of the service composition model is that by defining atomic and composite groups of an eBusiness process and using compensation, a contingency and a DE-rollback at those groups an execution failure of the eBusiness process can be automatically recovered at any composition level.

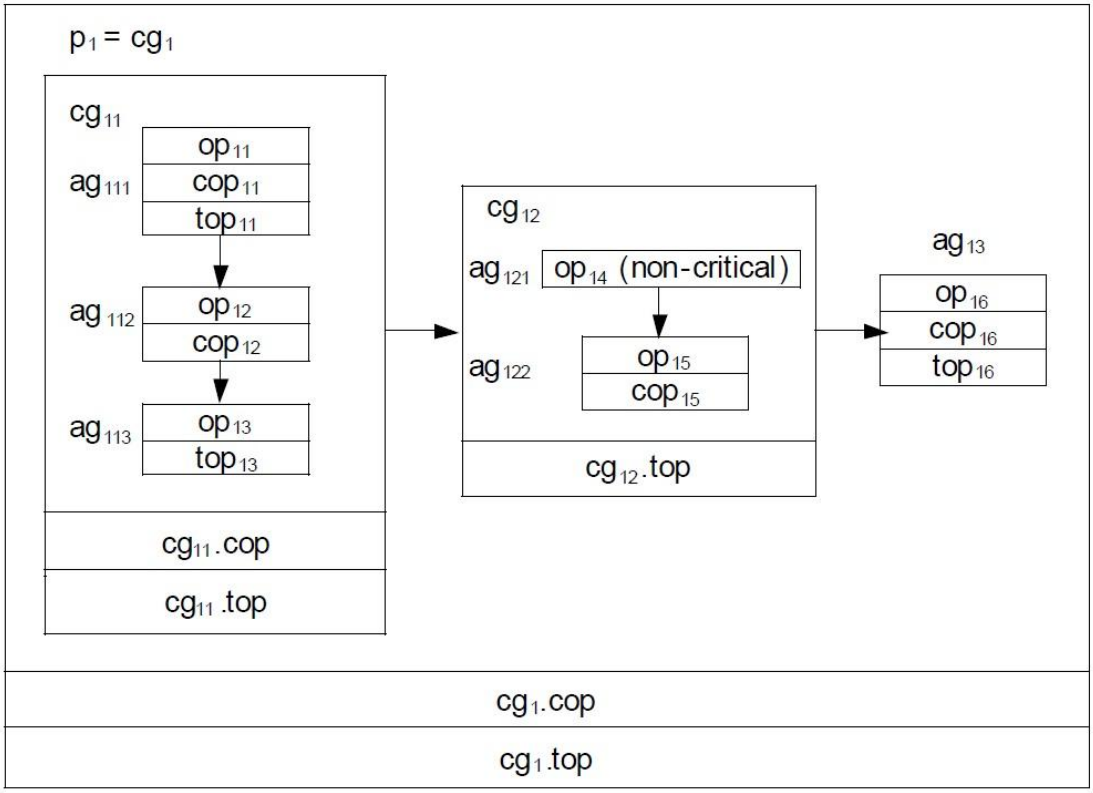

Figure 4. An abstract view of a sample process using the DeltaGrid service composition structure [UX09].

3.3 Execution Semantics and Recoverability of an Operation

The definitions for an ACID DEGS operation and a multilevel DEGS operation are given and their execution and its influence to an operation are described. A pre-commit recoverability mechanism (subsection 3.3.1.1) and a post-commit recoverability mechanism (subsection 3.3.1.2) of a DEGS operation are introduced. An execution of an atomic group (subsection 3.3.2) and a composite group (subsection 3.3.3) are explained as well. A criticality attribute is defined (subsection 3.3.2) and the terms a shallow compensation and a deep compensation are extended compared to their original definition (subsection 3.3.3). A backward recovery of the atomic group and the composite group are explained (subsection 3.3.4).

An operation was defined as a single activation of a service of a process. In the DeltaGrid environment the operation is an activation of a DEGS service which is an autonomous entity taking care of its local correctness using a concurrency control mechanism [UX09]. In the service composition model, the operation is a part of an atomic or a composite group. In practice, it means that DEGS service affects to their state in case of the backward recovery because the failure of the operation or because another operation's failure in the composite group execution. A functionality and an implementation of the DEGS service varies depending on the provider of the DEGS service, which is why an operation can be an ACID DEGS operation or a multilevel DEGS operation [UX09]. An *ACID DEGS operation* has a transaction, which can automatically do rollback by underlying data base if the operation fails. *A multilevel DEGS operation* has several sub-transactions which are like the transaction of the ACID DEGS operation and which can commit unilaterally. If one of the sub-transactions fails, the rollback could not be done because some other sub-transactions might have been committed. Then a local compensating transaction will be run for that the operation will reach a consistent state. The *local compensating transaction* is atomic.

3.3.1 Execution Semantic of a DEGS Operation

An operation can be an ACID DEGS operation or a multilevel DEGS operation that affects to the execution of it. The differences between those operation types are the number of the termination states, the ACID DEGS operation never terminates in the failed state and a multilevel DEGS operation can execute a compensation to be able to commit the operation after the execution of the operation have at first failed. Figure 5 shows the semantics of a

transaction of a DEGS operation [UX09]: the states and the actions between the states of a) an ACID DEGS operation and b) a multilevel DEGS operation. When a DEGS operation has been invoked it enters to the active state. If the execution of the DEGS operation succeeds the state of the DEGS operation changes to the successful but if the execution fails the state of the DEGS operation will become failed. An underlying database system of the ACID DEGS operation supports a rollback and it is run automatically. The rollback is supposed to finish successfully every time [UX09]. After that the state of the ACID DEGS operation is aborted. So the termination states of the ACID DEGS operation are successful and aborted.

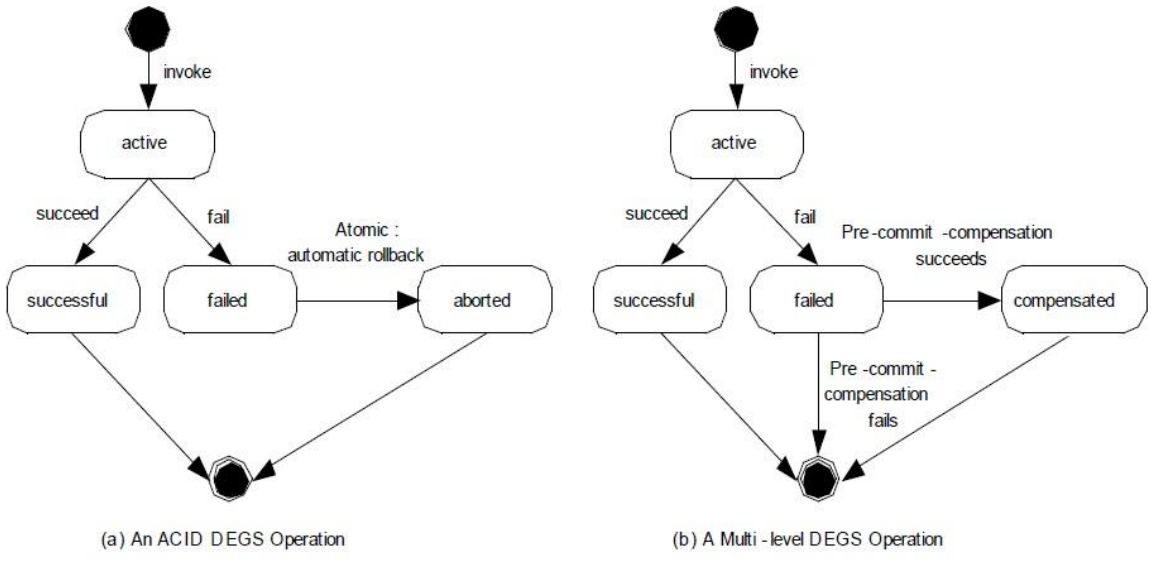

Figure 5. Transaction semantics of a DEGS operation [UX09].

After a multilevel DEGS operation has failed, a local compensating transaction is activated automatically by DEGS service and not by the DeltaGrid recovery capability [UX09]. Because the multilevel DEGS operation may have several sub-transactions, there might be a need to run many compensation steps. The compensating transaction is run before committing the whole DEGS operation as a multi-level transaction [UX09]. It is referred to as *pre-commit-compensation* which leads to a state compensated if it is run successfully. If the pre-commit-compensation fails a *pre-commit recoverability* mechanism cleans the consequences of the tried pre-commit-compensation and the situation is as before starting to run the pre-commit-compensation, because the pre-commit-compensation is an atomic transaction [UX09]. The state of the multilevel DEGS operation remains failed. The termination states of the multilevel DEGS operation are successful, failed and

compensated. So the multi-level DEGS offers a failure recovery mechanism called pre-commit-compensation.

3.3.1.1 Pre-commit Recoverability

A pre-commit recoverability mechanism of a DEGS operation is needed to clean up the consequences of the tried pre-commit-compensation to revert to the situation as before starting to run the pre-commit-compensation. There are four options to do the cleaning and they are listed in Table 4 [UX09]. When a DEGS operation performs pre-commit recoverability one of the following subsequent is selected: an automatic rollback, a pre-commit compensation, a DE-rollback or a service reset function.

Option	Meaning
Automatic rollback	The failed service execution can be automatically rolled back by a service provider.
Pre-commit-compensation	A pre-commit-compensation is invoked by a service provider to backward recover a failed operation.
DE-rollback	A failed operation can be reversed by executing DE-rollback.
Service reset	The service provider offers a service reset function to clean up the service execution environment.

Table 4. Pre-commit recoverability options of a DEGS operation [UX09].

An ACID DEGS operation will run automatic rollback, which means that a service provider runs a rollback for the failed service [UX09]. In case of a multi-level DEGS, the service provider runs a pre-commit-compensation [UX09]. If the pre-commit-compensation fails and the semantic recovery condition is fulfilled, a DE-rollback will be run [UX09]. If the semantic recovery condition is not satisfied, *service reset* will be performed [UX09]. Then the service provider offers a function that cleans up the execution environment of the failed

service. To execute the service reset, usually a special program or a human agent is required [UX09].

The state diagram of the DEGS operation taking into account the pre-commit-compensation options is illustrated in Figure 6 [UX09]. If an ACID or a multi-level DEGS operation succeeds to run it, moves from active state to successful. If the ACID DEGS operation fails, the state will be aborted. In case of the multi-level DEGS operation fails, the state will be failed. Then the pre-commit-compensation will be performed. If it succeeds, the state becomes compensated and if it fails, the state stays failed. The DeltaGrid system will check the semantic recovery conditions and based on them, it initiates either DE-rollback or service-reset [UX09]. If they are fulfilled, a DE-rollback will be run and the state moves to DE-rollback. If the semantic recovery conditions are not satisfied, service reset will be performed and the state moves to service-reset. The execution of a pre-commit recoverability option: automatic rollback, pre-commit-compensation, DE-rollback or service reset moves a failed execution of DEGS operation to the state *pre-commit recovered* which represents one of the states: aborted, compensated, DE-rollback and service-reset. The final states of the DEGS operation are successful or pre-commit recovered. That is, one state less compared to the transaction semantics of a multi-level DEGS operation in Figure 5. The state failed is missing. So the pre-commit recoverability options increase the level of the consistency of the multi-level DEGS operation as all pre-commit recoverability options clean up the service execution environment of the failed multi-level DEGS operation.

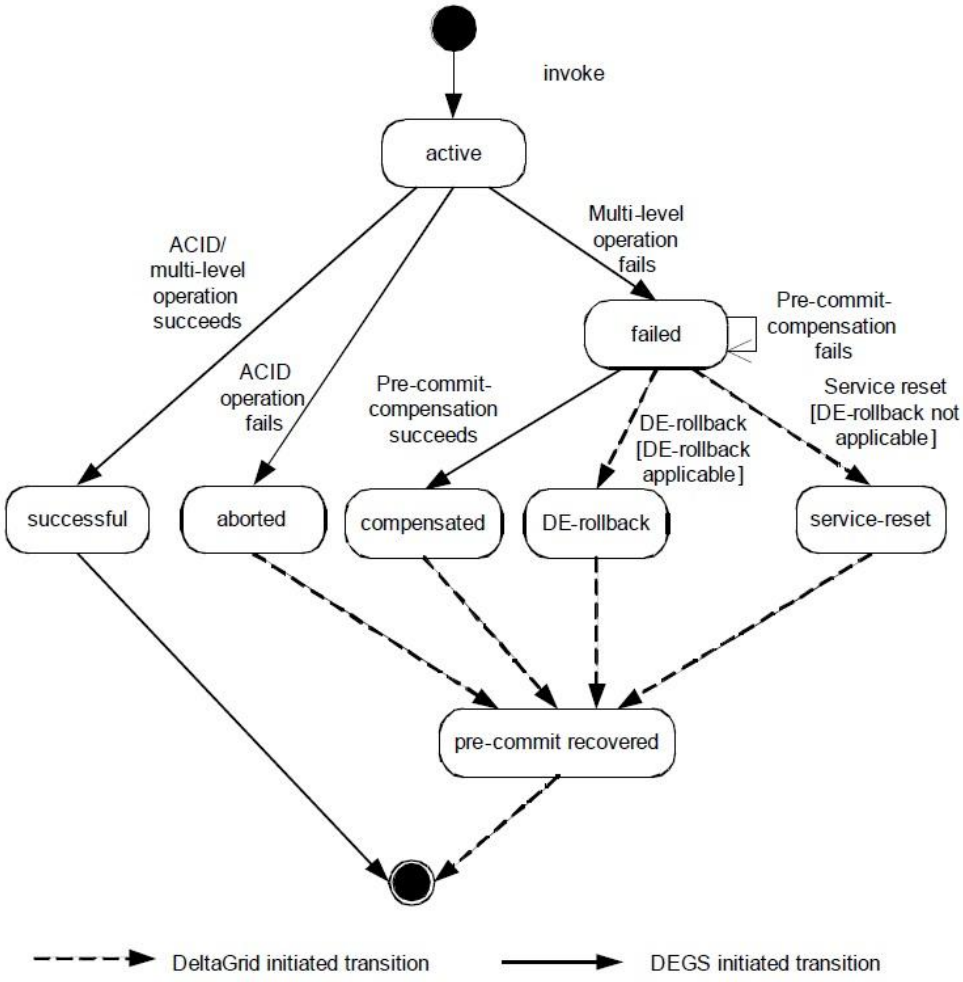

Figure 6. A DEGS operation taking into account the pre-commit-compensation options [UX09].

3.3.1.2 Post-commit Recoverability

A *post-commit recoverability* mechanism of a DEGS operation is needed to semantically undo a successfully terminated operation because there has been a failure in the execution of another operation [UX09]. There are three options to undo the effects of the successfully terminated operation and they are listed in Table 5 [UX09]. *Reversible* means that the data values, which the successfully completed operation has changed will be reversed. *Compensatable* option runs an operation called *post-execution compensation* to semantically undo the successfully completed operation. *Dismissible* means that there is no need to do any data cleaning up after a successfully terminated operation in the post-commit recovery situation.

Option	Meaning
Reversible (DE-rollback)	A completed operation can be undone by reversing the data values that have been modified by the operation execution.
Compensatable	A completed operation can be semantically undone by executing another operation, referred to as post-execution compensation.
Dismissible	A completed operation does not need any cleanup activities.

Table 5. Post-commit recoverability options of a DEGS operation [UX09].

The condition to start the post-commit recovery later referred as *post-commit recovery condition* is introduced in Subsection 3.3.3 . It takes effect only when a component of the composite group is a successfully terminated operation which needs to be semantically undone [UX09].

3.3.2 Execution Semantic of an Atomic Group

An atomic group has an optional compensation operation and on the top of that it may have a contingency operation as well which improves the ability of the atomic group to execute an operation successfully. Due to the relaxed atomicity, it is not necessary that all the operations of the composite group must have run successfully to be able to terminate the process successfully. For that, there is a *criticality attribute* of an atomic group, which is *critical* if an atomic group must be executed successfully to be able to continue the execution of the composite group, which the atomic group belongs to [UX09]. If the composite group can continue execution regardless of the failure of its atomic group, the atomic group is marked *non-critical* [UX09]. A default option is that an atomic group is critical [UX09]. If the critical atomic group fails, a contingency of the atomic group will be run [UX09]. If the atomic group is non-critical, the contingency is not needed. E.g., an atomic group ag_{113} is critical in Figure 4. If an operation op_{13} of the atomic group ag_{113} fails then a contingency top_{13} will be run. An atomic group ag_{121} is non-critical. If the operation

of it op_{14} fails then the composite group cg_{12} can continue by running an atomic group ag_{122}.

The execution semantics of an atomic group ag is depicted in Figure 7 [UX09]. When the primary operation is started, the ag becomes active. If the primary operation succeeds, the ag moves to the state successful. If the primary operation fails, ag moves to the state pre-commit recovered as described in Figure 6. If the atomic group is critical, the DeltaGrid system will start a contingency of the atomic group. If it is runs successfully, ag move to the state *ag successful*. The contingency is an atomic transaction of the DEGS operation similar to a pre-commit-compensation [UX09]. If the contingency fails, ag moves to *ag aborted* state. If the atomic group is non-critical, the ag moves from the state pre-commit recovered to the state ag aborted without performing contingency. The termination states of the atomic group are ag successful or ag aborted.

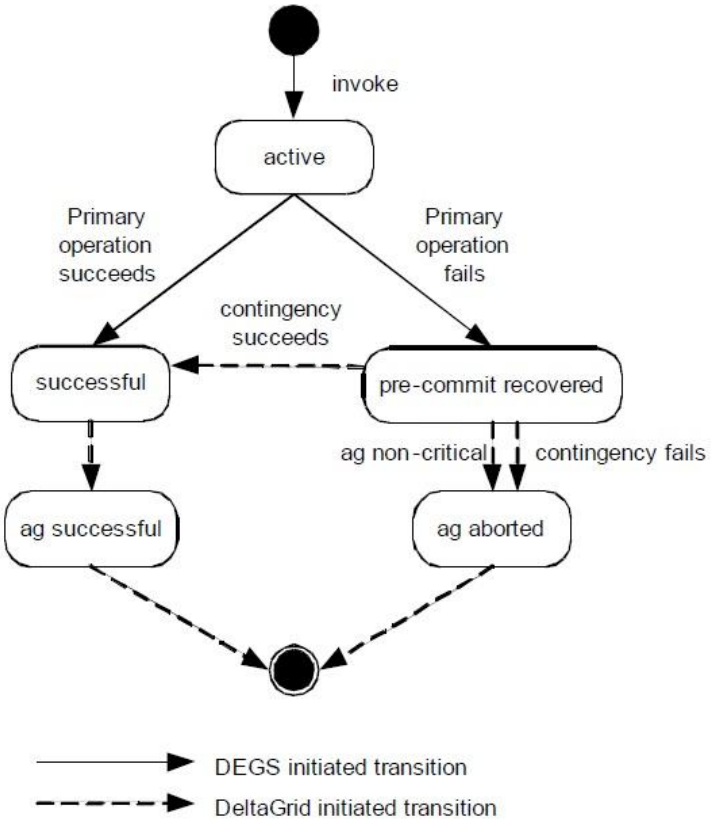

Figure 7. Execution semantics of an atomic group [UX09].

3.3.3 Execution Semantic of a Composite Group

Before introducing the execution semantic of a composite group, two terms shallow compensation and deep compensation will be extended compared to their original definition [LA95]. A composite group cg_{ik} is defined cg_{ik}= <($ag_{i,k,m}$ | $cg_{i,k,n}$)$^{+}$ [,cop_{ik}] [,top_{ik}]> in Table 3 [UX09]. A *shallow compensation* of the composite group cg_{ik} is the activation of the compensation operation cop_{ik} [UX09]. A composite group cg_{ik} can be composed of subgroups which is an atomic group ag_{ijm}=<op_{ij} [,cop_{ij}] [,top_{ij}]> or a composite group cg_{ikn}= <($ag_{i,k,n,x}$ | $cg_{i,k,n,y}$)$^{+}$ [,cop_{ikn}] [,top_{ikn}]> [UX09]. A *deep compensation* of the composite group cg_{ik} is the activation of the post-commit recovery either compensation or DE-rollback for each executed subgroup of the composite group: cop_{ij} for an atomic group and cop_{ikn} for a nested composite group [UX09]. But there is a case according to the post-commit recovery condition when a pre-commit recovery is run instead of the post-commit recovery. *A post-commit recovery condition* defines that if the failed subgroup is the first subgroup of the enclosing composite group, the subgroup will run a pre-commit-recovery. In all other cases all preceding subgroups will run a post-commit recovery. A shallow compensation is needed when a composite group has terminated successfully but its effects need to semantically undo because another operation has failed [UX09]. The deep compensation is needed when a failure of the subgroup causes the composite group to fail [UX09]. Then post-commit recovery needs to be started for all performed subgroups [UX09]. The deep compensation is also needed when a composite group terminates successfully and a compensation is needed, but the composite group does not have a shallow compensation [UX09].

An execution semantic of a composite group cg_i, which is composed of atomic groups cg_i= <$ag_{i,k}$$^{+}$ [,cop_i] [,top_i]> is depicted in Figure 8a [UX09]. The composite group cg_i is in a state active as long as its subgroups are performed. If all the subgroups succeed, cg_i moves to a state *cg_i successful*. If a subgroup ag_{ik} fails, it tries to run contingency top_i but if it also fails then cg_i moves to *ag_{ik} aborted.* If ag_{ik} is the first subgroup of cg_i, pre-commit recovery of ag_{ik} moves cg_i to a state *cg_i aborted*. In all other cases the executed subgroups ($ag_{i,1, \ldots, k-1}$) will perform post-commit recovery successfully moving cg_i to a state *cg_i deep compensated.* The post-commit recovery techniques of the atomic group of the composite group are DE-rollback and compensation [UX09]. A state *cg_i extended abort* represents one of the states: cg_i aborted or cg_i deep compensated. The state cg_i extended abort means that the partial results of the execution of the composite group cg_i have been

cleaned up and the composite group cg_i is ready for a contingency [UX09] top_i. If the contingency of the composite group succeeds, the cg_i moves to the state cg_i successful. If the contingency fails it will be rolled back as an atomic transaction and the cg_i stays in the state cg_i extended abort [UX09]. When the deep compensation is performed by compensating all the executed subgroups $ag_{i,1, \ldots, k-1}$ of the cg_i, all of them must succeed to move cg_i to the state cg_i deep compensated. If a compensation of a subgroup fails, a DE-rollback or a service reset function will be performed in a similar way as the failure of the pre-commit-compensation is handled. The first recovery option is DE-rollback and it is performed if the semantic recovery condition holds [UX09]. If not then the second option is the service reset function [UX09].

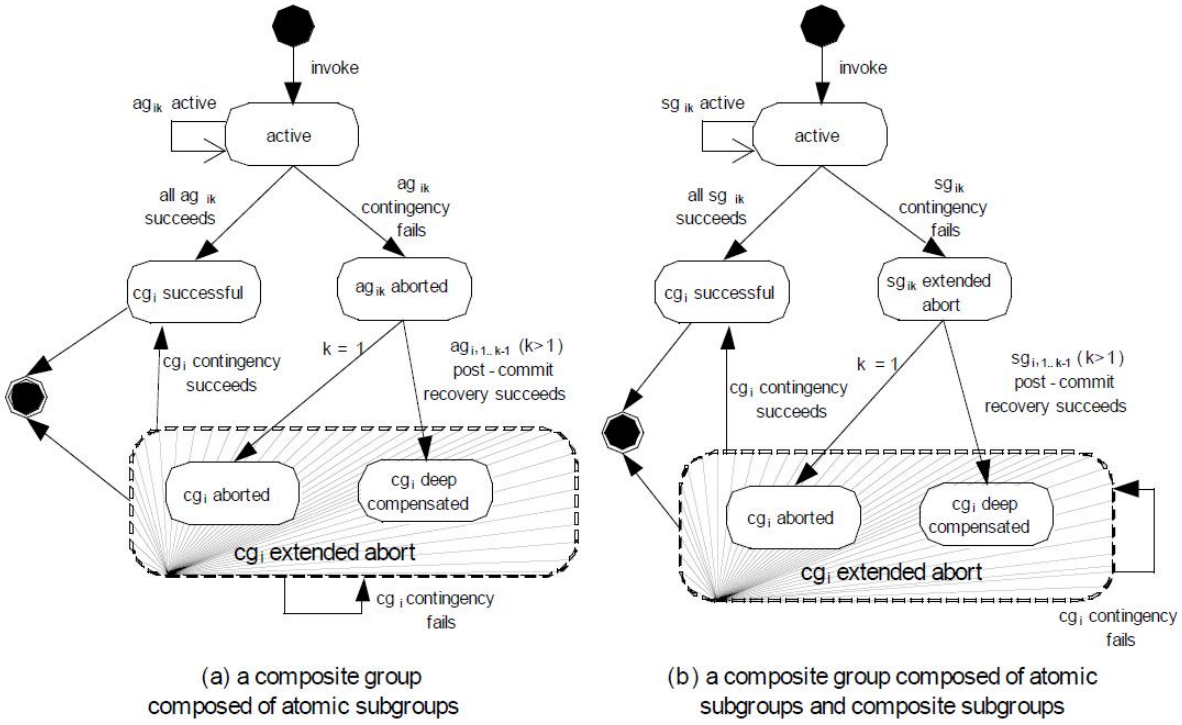

Figure 8. Execution semantics of a composite group [UX09].

An execution semantic of a composite group cg_i, which is composed of subgroups $sg_{i,k}$ is depicted in Figure 8b [UX09]. A subgroup $sg_{i,k}$ can be an atomic group or a composite group $cg_i = <sg_{i,k}^{+} [,cop_i] [,top_i]>$ [UX09]. The execution of the composite group, which is composed of subgroups, resembles the execution of the composite group composed of atomic groups [UX09]. The composite group cg_i is in a state active during the execution of

its subgroups. If all the subgroups succeed, cg_i moves to a state cg_i successful. If an atomic subgroup fails, it tries to run contingency top_i but if the latter also fails then cg_i moves to *sg_{ik} extended abort* as described in Figure 8a. If the composite subgroup fails, it runs contingency top_{ij} but if it also fails then cg_i moves to sg_{ik} extended abort as described in Figure 8b. If sg_{ik} is the first subgroup of cg_i pre-commit recovery of sg_{ik} is preformed and cg_i moves to a state cg_i aborted. In all other cases, the executed subgroups ($sg_{i,1, ..., k-1}$) will perform post-commit recovery successfully moving cg_i to a state cg_i deep compensated. If the contingency of the composite group top_i succeeds, the cg_i moves to the state cg_i successful as happened in Figure 8a. If the contingency fails it will be rolled back and the cg_i stays in the state cg_i extended abort as happened in Figure 8a. When the deep compensation is performed by compensating all the executed subgroups $sg_{i,1, ..., k-1}$ of the cg_i, all of the subgroups must succeed to move cg_i to the state cg_i deep compensated [UX09] as happened to the composite group, which is composed of atomic groups in Figure 8a . In both cases the composite group consisting of the atomic groups or the subgroups, the termination states are cg_i successful or cg_i extended abort. So the final state of the composite group is cg_i successful or cg_i extended abort, which means that the execution of the composite group is never left to an inconsistent state.

3.3.4 Backward Recovery of an Atomic Group and a Composite Group

A successfully terminated atomic group may need to backward recover due to the failure of another entity of the process [UX09]. The backward recovery cleans up all the effects of the successfully terminated atomic group ag. The successfully terminated atomic group is depicted in Figure 7 in a final state ag successful. Figure 9 [UX09] continues the state diagram of the atomic group in the case the atomic group needs to backward recover. The backward recovery of the atomic group starts in the state ag successful in Figure 9. When the backward recovery is needed, a recovery option of the atomic group is activated based on post-commit recoverability of the primary operation [UX09]. The post-commit recoverability of the primary operation is depicted in Figure 6, where a multi-level operation fails and pre-commit compensation is tried by compensation, DE-rollback or service-reset. The backward recovery options of the atomic group are compensation, DE-rollback and service-reset. The listing order of the options is also the performance order of the backward recovery options of the atomic group when trying to run different recovery options. If the primary operation (DEGS) of the atomic group is compensatable (defined in Table 5) the compensation cop_{ij} will be performed. If it succeeds, the atomic group moves

to the state *ag compensated*. If the compensation fails, the atomic group stays in the state ag successful until the effect of the tried compensation have been cleaned up by DEGS execution environment. Then the DE-rollback will be performed if the atomic group is DE-rollback applicable. It means that the primary operation of the atomic group (DEGS) is reversible (defined in Table 5). After the DE-rollback, the atomic group move to the state *ag DE-rollback*. If the atomic group is not reversible, the service reset (defined in Table 4) will be run and the atomic group moves to the state *ag service-reset*. There is an error in Figure 9: the guard condition of the service reset transition should be *not DE-rollback applicable* instead of DE-rollback applicable, because the backward recovery option service reset is tried after the DE-rollback option. All the three states: ag compensated, ag DE-rollback and ag service-reset move the atomic group to the state *ag post-commit recovered,* which represents one of them.

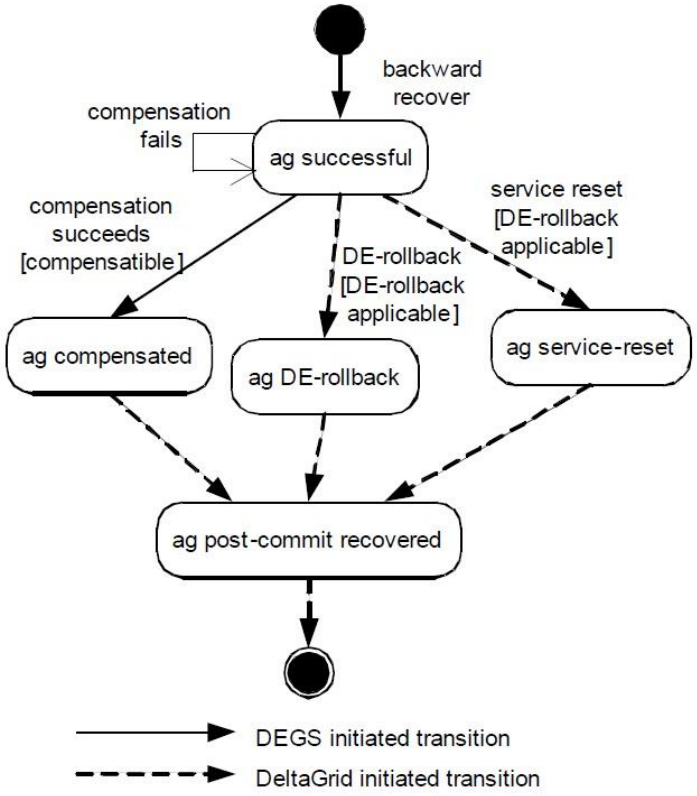

Figure 9. Backward recovery semantics of an atomic group [UX09] Note: The guard condition of the service reset transition should be *not DE-rollback applicable* instead of DE-rollback applicable.

A successfully terminated composite group may need to backward recover due to the failure of another entity of the process. The backward recovery cleans up all the effects of the successfully terminated composite group cg_i. In a backward recovery of a successfully terminated composite group, a shallow compensation is preferred to a deep compensation [UX09]. E.g., when a critical subgroup ag_{13} of the composite group cg_1 in Figure 4 has failed both op_{16} and top_{16}, the successfully performed composite groups cg_{11} and cg_{12} will be compensated. Because cg_{11} has a shallow compensation cg_{11}.cop it will be performed but because cg_{12} does not have a shallow compensation a deep compensation cg_{12}.top will be performed. The successfully terminated composite group is depicted in Figure 8 in a final state cg_i successful. Figure 10 [UX09] continues the state diagram of the composite group in the case the composite group needs to backward recover.

The backward recovery of the composite group cg_i, which is composed of atomic groups starts in the state cg_i successful in Figure 10 a [UX09]. The composite group cg_i runs the shallow compensation by activating the compensation operation cop_i if the shallow compensation is available. If the shallow compensation succeeds, the composite group moves to the state *cg_i shallow compensated*. If the shallow compensation fails, the composite group stays in the state cg_i successful and starts the deep compensation. If the shallow compensation is not available meaning the composite group does not have a shallow compensation, the composite group starts straight away a deep compensation. The deep compensation is performed by running a backward recovery for every atomic group ag_{ik} of the composite group cg_i [UX09]. The only termination state of the backward recovery of the atomic group ag is ag post-commit recovered as shown in Figure 9. It ensures that when all the atomic groups ag_{ik} have executed the post-commit recovery, the cg_i moves to the state *ag_{ik} posti-commit recovered*. It causes that cg_i becomes *cg_i deep compensated*.

A backward recovery of a composite group cg_i, which is composed of subgroups $sg_{i,k}$, starts in the state cg_i successful in Figure 10 b [UX09]. A subgroup $sg_{i,k}$ can be an atomic group or a composite group. The backward recovery of the composite group, which is composed of subgroups, resembles the backward recovery of the composite group, which is composed of atomic groups. The composite group cg_i runs the shallow compensation if the shallow compensation is available. If the shallow compensation succeeds, the composite group moves to the state cg_i shallow compensated. If the shallow compensation fails the composite group stays in the state cg_i successful and starts the deep

compensation. If the shallow compensation of the cg_i does not exist the composite group starts straight away a deep compensation. The deep compensation is performed by running a backward recovery for every subgroup sg_{ik} of the composite group cg_i [UX09]. Depending on the subgroup sg_{ik} the backward recovery is done in different ways. If the subgroup sg_{ik} is an atomic group the post-commit recovery is performed and the subgroup sg_{ik} terminates in the state *sg_{ik} post-commit recovered*. If the subgroup sg_{ik} is a composite group of the subgroups, it terminates either in the state *sg_{ik} shallow compensated* (if the shallow compensation is available and succeeded) or *sg_{ik} deep compensated* (if the shallow compensation not available or did not succeed) as shown in Figure 10 b according to the shallow and deep compensation rules of the composite group. Then all the subgroups sg_{ik} of the composition group are backward recovered and the composite group cg_i becomes *cg_i deep compensated*. So all the states: sg_{ik} post-commit recovered, sg_{ik} shallow compensated and sg_{ik} deep compensated move the composite group to the state cg_i deep compensated, which represents one of the states. So the final state of the backward recovery of the composite group is cg_i shallow compensated or cg_i deep compensated, which means that the backward recovery of the composite group is never left to an inconsistent state.

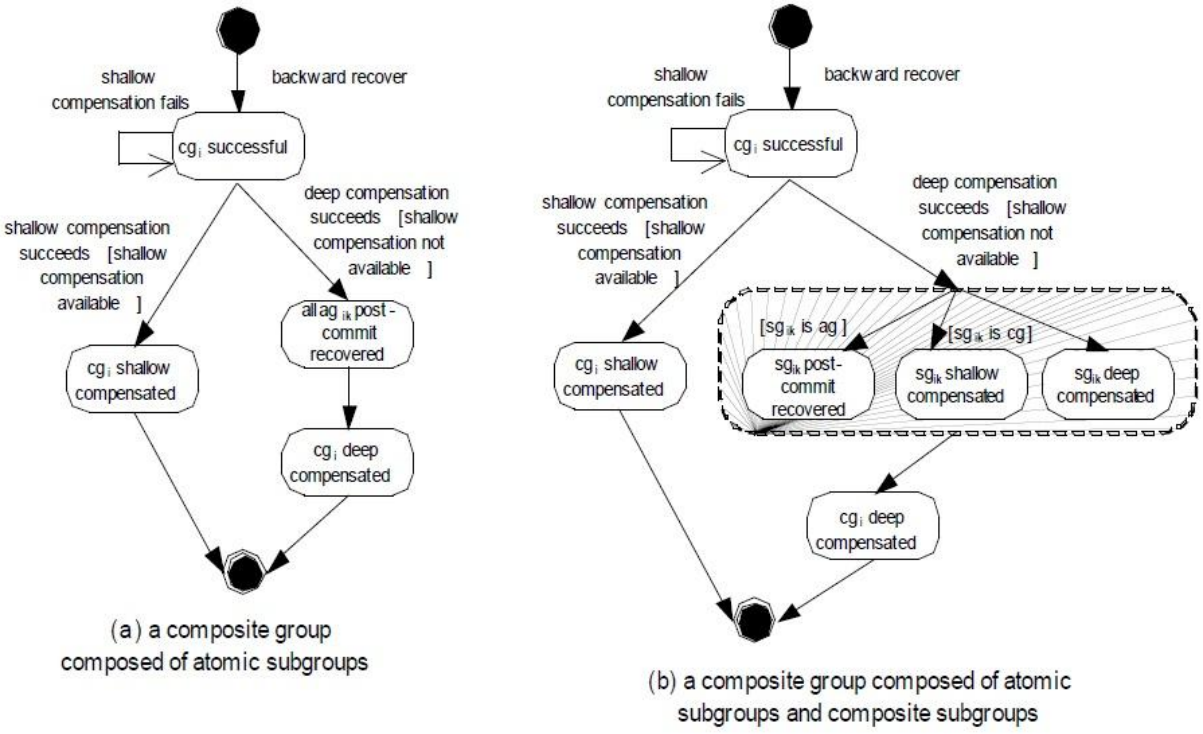

Figure 10. Backward recovery semantics of a composite group [UX09].

3.4 Recovery Algorithms

The following algorithms are presented and explained: to make a choice between DE-rollback or service reset for an operation (subsection 3.4), a post-commit recovery for an atomic group (subsection 3.4.1), deep compensation for a composite group(subsection 3.4.2), a recovery of an operation from a failure in the context of a running process (subsection 3.4.3) and also for the atomic group to propagate a failure (subsection 3.4.3). The conditions for the applicability of the different recovery options are defined and the failure recovery algorithm of the operation is demonstrated using an example (subsection 3.4.3).

3.4.1 Post-commit Recovery Algorithm of an Atomic Group

An atomic group ag is *complete* if it has required compensation and contingency plans for the primary operation of the atomic group [UX09]. The need of the compensation and the contingency plan depends on the post-commit recoverability of the primary operation and the value of the criticality attribute of the atomic group [UX09]. A complete critical atomic group has a contingency plan where as a complete non-critical atomic group has only the primary operation. The post-commit recoverability defines if a complete critical atomic group must have a compensation plan [UX09]. If the primary operation is compensatable, the atomic group must have a compensation plan whereas a reversible or a dismissible primary operation does not need it. In practice, there is a risk that a specification of the service provider of the atomic group does not include the compensation plan even if the definition of the process requires compensation [UX09]. In this circumstance, the algorithm called *DE-rollbackOrServiceReset(Operation* op_{ij}*)* decides either to activate DE-rollback or service reset on the operation [UX09]. The decision depends on if the semantic recovery condition of the DE-rollback is satisfied or not. The algorithm of DE-rollbackOrServiceReset(Operation op_{ij}) is shown in Figure 11 [UX09]. It describes how the primary operation op_{ij} recovers through DE-rollback or service reset. The input of the algorithm is a failed operation, which should backward recover [UX09]. The result of the algorithm is that the effects of the failed operation have been cleaned up in the service execution environment. The algorithm checks if the semantic recovery condition holds. If yes then the DE-rollback will be activated. If not then the DEGS operation activates service reset. An algorithm called post-commitRecoverAtomicGroup(AtomicGroup ag_{ij}) uses this algorithm when compensation is not available or when compensation fails [UX09].

```
public void DE-rollbackOrServiceReset(Operation opij)
{
        //check if DE-rollback dopij is applicable
        CASE:
                1. dopij APPLICABLE
                        EXECUTE dopij;
                        RETURN;
                2. dopij NOT APPLICABLE
                        EXECUTE service reset;
                        RETURN;
        END CASE;
}
```

Figure 11. Procedure to invoke DE-rollback or service reset on an operation [UX09].

The atomic group post-commit recovery algorithm called *post-commitRecoverAtomicGroup(AtomicGroup ag_{ij})* is written in Figure 12 [UX09]. It defines a backward recovery of an atomic group after the atomic group has terminated successfully based on the value of the criticality attribute of the atomic group and on the post-commit recoverability of the primary operation. The input of the algorithm is an atomic group, which should post-commit recover. The result of the algorithm is that the atomic group has post-commit recovered using compensation, DE-rollback or service rest.

The algorithm describes more in detail a backward recovery semantics of the atomic group depicted in Figure 9. The algorithm starts in the state ag_{ij} successful. The successful compensation (case 1.1.1) moves ag_{ij} to the state ag_{ij} compensated. If the compensation fails (case 1.1.2), compensation not defined (case 1.2) or compensation not necessary (case2), DE-rollback or service rest is activated which moves ag_{ij} to the state ag_{ij} DE-rollback or ag_{ij} service-rest. If the ag_{ij} does not need to backward recover (case 3), it is dismissible as defined in Table 5. The algorithm ensures that ag_{ij} will be in the state ag_{ij} post-commit recovery if a backward recovery is needed. The algorithm checks the post-commit recoverability of the primary operation op_{ij} of the given atomic group ag_{ij}. Based on it, a case is chosen:

1. op_{ij} is compensatable. If ag_{ij} has compensation cop_{ij}, cop_{ij} will be activated. If it succeeds, ag_{ij} is compensated and the algorithm returns. If cop_{ij} fails, the algorithm will start DE-rollbackOrServiceReset(op_{ij}) written in Figure 11.

2. op_{ij} is reversible, which means that ag_{ij} does not have cop_{ij}. The algorithm will start DE-rollbackOrServiceReset(op_{ij}).

3. op_{ij} is dismissible, which means that ag_{ij} does not need to backward recover. So no action is needed.

```
public void post -commitRecoverAtomicGroup (AtomicGroup agij)
{
    get agij 's primary operation => opij;
    //check opij's post -commit recoverability
    CASE:
        1. opij is COMPENSATABLE :
            //check if agij has compensation copij
            CASE:
                1.1 agij has copij:
                    EXECUTE copij;
                    //check copij execution result
                    CASE:
                        1.1.1 copij SUCCEEDS :
                            RETURN;
                        1.1.2 copij FAILS:
                            EXECUTE DE-rollbackOrServiceReset (opij);
                            RETURN;
                    END CASE ;
                1.2 agij has no copij:
                    EXECUTE DE-rollbackOrServiceReset (opij);
                    RETURN;
            END CASE ;
        2. opij is REVERSIBLE :
            EXECUTE DE-rollbackOrServiceReset (opij);
            RETURN;
        3. opij is DISMISSIBLE :
            RETURN;
    END CASE ;
}
```

Figure 12. Atomic group post-commit recovery algorithm [UX09].

3.4.2 A Deep Compensation Algorithm of a Composite Group

A deep compensation of a composite group will be executed if a) a critical subgroup of it fails before the composite group completes or b) the composite group completes

successfully, but does not have a shallow compensation although it needs to backward recover because another operation outside of the composite group has failed. In both cases a) and b) the deep compensation includes the post-commit recovery of the executed critical subgroups. E.g., if the atomic group ag_{112} fails in Figure 4 [UX09], the atomic group ag_{111} must be compensated which is the deep compensation of the enclosing composite group cg_{11} before cg_{11} completes. If the atomic group ag_{13} fails, the composite groups cg_{12} and cg_{11} will be compensated which is the deep compensation of cg_1 before cg_1 completes. The composite group cg_{12} must be deep compensated by running ag_{122} because cg_{12} does not have a shallow compensation. cg_{11} will be shallow compensated by running cg_{11}.cop.

An algorithm of the deep compensation of the composite group called *deepCompensate(CompositeGroup* cg_i*)* is written in Figure 13 [UX09]. It recursively activates the deep compensation of the enclosing composite group of a subgroup if the contingency of the subgroup fails. The input of the algorithm is the composite group, which needs to deep compensate [UX09]. The result of the algorithm is that the effects of the composite group have been semantically undone by running post-commit recovery of the completed subgroups in the reversed order, if a subgroup is critical for the composite group [UX09]. At first the algorithm gets a list of executed critical subgroups sg_{ik} of the composite group cg_i in the reversed execution order. Then the algorithm iterates through every subgroup sg_{ik} of cg_i. If sg_{ik} is an atomic group post-commitRecoverAtomicGroup(sg_{ik}) will be started. If sg_{ik} is a composite group and has shallow compensation csg_{ik} , it will be run. If sg_{ik} does not have shallow compensation or the shallow compensation fails, the deep compensation of sg_{ik} will be started recursively.

The algorithm describes more in detail a deep compensation of a composite group if it has completes successfully before the compensation depicted in Figure 10 b. The algorithm starts in the state cg_i successful. If a subgroup sg_{ik} is an atomic group (case 1), sg_{ik} moves to the state sg_{ik} post-commit recovered, by running the post-commit recovery algorithm for sg_{ik}. If sg_{ik} is a composite group and it has shallow compensation (case 2.1.1), a successfully terminated shallow compensation moves sg_{ik} to the state sg_{ik} shallow compensated. It the shallow compensation fails (case 2.1.2) or sg_{ik} does not have shallow compensation (case 2.2) the deep compensation of sg_{ik} will be run. It moves sg_{ik} to the state sg_{ik} deep compensated.

The algorithm describes also more in detail a deep compensation of a composite group if a critical subgroup of it fails before the composite group completes. It is depicted in Figure 8 b. The algorithm starts in the state sg_{ik} extended abort. Then all the executed subgroups will be backward recovered and the cg_i moves to the state cg_i extended abort. So the algorithm confirms that composite group cg_i reaches the state cg_i extended abort if a critical subgroup of cg_i fails during the execution of the cg_i as shown in Figure 8 b and the state cg_i deep compensated is reached, when cg_i has completed successfully before the compensation and a deep compensation is needed for the composite group cg_i as shown in Figure 10 b.

```
public void deepCompensate   (CompositeGroup   cgi)
{
        //get a list of executed critical subgroups of      cgi in reverse execution order
        C = [sgik | sgik ∈ cgi] (k = n ..1)

        //iterate through every executed subgroup of      cgi
        FOR EACH  sgik ∈C
        {
                //check if  sgik is an atomic group
                CASE:
                        1. sgik is an  atomic group
                                EXECUTE  post-commitRecoverAtomicGroup  (sgik);
                                CONTINUE ;
                        2. sgik is a composite group :
                                //check if  sgik has shallow compensation     csgik
                                CASE:
                                        2.1 sgik has csgk:
                                                EXECUTE  csgk ;
                                                //check csgik execution result
                                                CASE:
                                                        2.1.1 csgik SUCCEEDS :
                                                                CONTINUE ;
                                                        2.1.2 csgik FAILS:
                                                                EXECUTE  deepCompensate (sgik);
                                                END CASE ;
                                        2.2 sgik has no  csgk:
                                                EXECUTE  deepCompensate (sgik) ;
                                                CONTINUE ;
                                END CASE ;
                END CASE ;
        }//END FOR ;
}
```

Figure 13. Composite group deep compensation algorithm [UX09].

3.4.3 A Recovery Algorithm of an Operation Execution Failure

A recovery algorithm for an execution failure of an operation in the context of a running process is called *recover(Operation op_{ij})* and written in Figure 14 [UX09]. The input of the algorithm is a failed operation op_{ij}. The result of the algorithm is a Boolean value depending on if the process of the failed operation op_{ij} can forward recover or not [UX09]. If the process can carry on the next execution entity it has been recovered and the algorithm returns the value true [UX09]. If the whole process have been backward recovered the algorithm returns the value false [UX09].

```
public boolean recover (Operation opij)
{
    get the enclosing atomic group of opij => agij;
    //check if agij is critical
    CASE:
        1. agij is non-critical:
            RETURN TRUE ;
        2. agij is critical:
            //check if agij has contingency topij
            CASE:
                2.1 agij has topij:
                    EXECUTE topij;
                    //check topij execution result
                    CASE:
                        2.1.1 topij SUCCEEDS :
                            RETURN TRUE ;
                        2.1.2 topij FAILS :
                            EXECUTE propogateFailure (agij);
                    END CASE ;
                2.2 agij has no topij:
                    EXECUTE propogateFailure (agij);
            END CASE ;
    END CASE ;
}
```

Figure 14. Operation failure recovery algorithm [UX09].

At first the algorithm gets an enclosing atomic group ag_{ij} of the failed operation op_{ij}. After checking that ag_{ij} is non-critical, the algorithm returns the value true. If the ag_{ij} is critical and it has a contingency top_{ij} the contingency will be started. When it has succeeded the algorithm returns the value true. If the contingency fails or ag_{ij} does not have the

contingency, the algorithm propagateFailure(ag_{ij}) will be started. The fault of the operation op_{ij} will be passed to the enclosing atomic group ag_{ij} by running *propagateFailure(ag_{ij}),* which will recover the failed atomic group in the context of enclosed nested composite group execution [UX09].

```
public boolean propagateFailure    (AtomicGroup   agij)
{
        get the enclosing composite group of      agij => cgi ;
        //check if cgi exists
        WHILE (cgi is not NULL  )
        {
                //check if cgi is critical
                CASE:
                        1. cgi is non-critical :
                                RETURN TRUE  ;
                        2. cgi is critical :
                                EXECUTE  deepCompensate (cg);
                                //check if cgi has contingency   topi
                                CASE:
                                        2.1 cgi  has topi:
                                                EXECUTE  topi;
                                                //check topi execution result
                                                CASE:
                                                        2.1.1 topi SUCCEEDS :
                                                                RETURN TRUE  ;
                                                        2.1.2 topi FAILS :
                                                                get the enclosing composite group of      cgi => cgi
                                                                CONTINUE ;
                                                        END CASE ;
                                        2.2 cgi  has no  topi:
                                                get the enclosing composite group of      cgi => cgi
                                                CONTINUE ;
                                END CASE ;
                CASE ;
        }//END WHILE  ;
        //the process has been backward recovered
        RETURN FALSE  ;
}
```

Figure 15. Atomic group failure propagation algorithm [UX09].

The algorithm propagateFailure(ag_{ij}) is written in Figure 15 [UX09]. The input of the algorithm is a failed atomic group ag_{ij}. The result of the algorithm is a Boolean value depending on if the process of the failed operation ag_{ij} can forward recover or not [UX09]. If the process has been recovered the algorithm returns the value true [UX09]. If the whole process have been backward recovered the algorithm returns the value false [UX09]. At first, the algorithm gets an enclosing composite group cg_{ij} of the failed atomic group ag_{ij}.

After checking that cg_{ij} is non-critical the algorithm returns the value true. If the cg_{ij} is critical the deep compensation is started. If cg_{ij} has a contingency top_{ij} it will be started. When it has succeeded the algorithm returns the value true. If the contingency fails or cg_{ij} does not have the contingency, the algorithm propagateFailure(cg_{ij}) will be started again [UX09]. The algorithm propagateFailure(cg_{ij}) will be called recursively. It will continue until either a) the contingency of the composite group succeeds or b) the composite group of the highest level (in practice the process) is reached [UX09]. In the case a), the process has successfully forward recovered and can carry on the next execution entity. In the case b), the whole process has backward recovered.

The recovery algorithm (recover(Operation op_{ij})) for an execution failure of an operation in the context of a running process, which is written in Figure 14 describes more in detail a failure recovery of an operation depicted in Figure 7. The algorithm starts in the state pre-commit recovered for operation op_{ij} [UX09]. If the enclosing atomic group ag_{ij} of the failed operation op_{ij} is non-critical (case 1), no contingency is needed and ag_{ij} moves to the state ag aborted. If the ag_{ij} is critical and contingency succeeds (case 2.1.1.) ag_{ij} moves to the state ag successful. In both cases, the value true is returned and the enclosing process carries on the next execution entity. If the contingency fails (case 2.1.2) or ag_{ij} does not have the contingency (case 2.2), the algorithm propagateFailure(ag_{ij}) will be started. The fault of ag_{ij} will be passed to the enclosing composite group cg_{ij} by running the algorithm propagateFailure(AtomicGroup ag_{ij}), which will recover the failed atomic group ag_{ij}. The algorithm starts at the state ag aborted in Figure 8 a. If an enclosing composite group cg_{ij} of ag_{ij} is non-critical (case 1), the algorithm returns the value true and the process will carry on the next execution operation. If the cg_{ij} is critical, the deep compensation is started and cg_i moves to the state cg_i extended abort. If cg_{ij} has a contingency top_{ij}, it will be started. When it has succeeded (case 2.1.1), cg_i moves to the state cg_i successful. The algorithm returns the value true and the enclosing process carries on the next execution operation. If the contingency fails (case 2.1.2) or cg_{ij} does not have the contingency (case 2.2), cg_i stays in the state cg_i extended abort. It is the state sg_{ij} extended abort in Figure 8 b. In that state the algorithm propagateFailure(cg_{ij}) will start the recursive call [UX09]. The failure of sg_{ij} starts the deep compensation of the enclosing composite group $cg_{ij.}$ If cg_{ij} has a contingency top_{ij} it will be started. When it has succeeded (case 2.1.1), cg_i moves to the state cg_i successful. The algorithm returns the value true and the enclosing process carries on the next execution operation. If the contingency fails (case 2.1.2) or cg_{ij} does not

have the contingency (case 2.2) the algorithm will be started recursively again. It will continue until cg_{ij} reaches the composite group of the highest level (in practice the process). If the highest level composite group has a contingency and it succeeds, the process moves to the state cg_i successful [UX09] which means that the process has successfully forward recovered and it can carry on to the next operation. In all other cases, the process moves on the state cg_i deep compensated and terminates [UX09], which means that the whole process has backward recovered. So the algorithm recover(Operation op_{ij})) combines the state transitions of the Figures 7, 8 and 10 [UX09] and fulfils them.

4 Constraint Condition and Logic Rule Based Recovery Mechanism

The essential terms e.g. a scope and a participant are defined and their relationship to an atomic service and a composite service are explained. An example of an eBusiness transaction using scopes is given (subsection 4.1). A completion constraint condition, an ignorable compensation constraint condition (subsection 4.2) and a logic rule (subsection 4.3) are explained and examples of their implementation are described. Possibility to combine constraint conditions and logic rules and use them together is explained. A management of the constraint rules is described briefly (subsection4.3).

A Constraint rules-based recovery mechanism is presented and explained (subsection 4.4). It makes use of four techniques: an atomic service retrying, a minimum range recovery, a synchronized compensation and a customer interaction. They are introduced and a phase when they are used in the Constraint rules-based recovery mechanism is depicted and described (subsection 4.4).

4.1 Scopes and Participants

The terms: a scope, a coordinator, a participant are defined and their relationship to an atomic service and a composite service are explained. An example of an eBusiness transaction using scopes is depicted and described in this subsection.

An eBusiness process can be managed with an eBusiness transaction which has a hierarchical structure described using scopes. A *scope* executes a certain sub task of the eBusiness transaction and it has a coordinator to manage its participants [CZM10]. A *participant* is a service provider or a consumer [CZM10]. The *coordinator* is a special participant which interacts with the participants of the scope. A sample eBusiness transaction workflow using scopes is pictured in Figure 4A. The rectangle is a symbol of atomic service in Figure 4A with exceptions of rectangles with Service request and End. A service provider is used to identify a service. P_a is an atomic service provided by service provider a in Figure 4A. A scope defines a composite service [CZM10] which is marked with a dashed rectangle in Figure 4A. SC_1 is a coordinator of the scope 1 in it. There are two atomic services and six scopes in Figure 4A . One of the scopes has two atomic

services and the others have three atomic services. A participant of the scope can be a sub coordinator as well [CZM10]. This way a scope offers a nested structure.

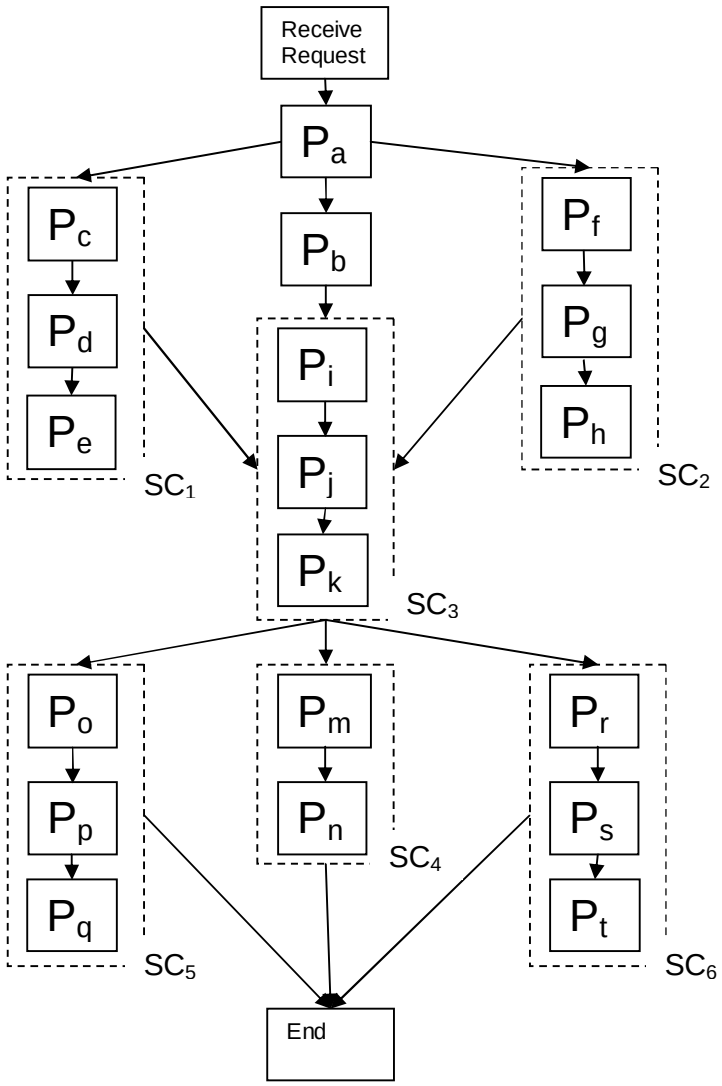

Figure 4A. A sample eBusiness transaction workflow using scopes.

4.2 Constraint Condition

Two kinds of constraint conditions exist: a completion constraint condition and an ignorable compensation constraint condition. Their usage is explained and examples of their implementation are given in this subsection.

The structure of an eBusiness process creates a structure for its transaction, which consists of the composite services, atomic services and relationships between them. Two types of constraint rules can be used to represent those relationships: constraint condition

and logic rule [CZM10]. *Constraint condition* determines if a service has been performed in an acceptable way in a transaction of a certain application [CZM10]. There are two kind constraint conditions: a completion constraint condition and an ignorable compensation constraint condition. The idea of the *completion constraint condition* (CCC) is that if a service of a transaction will not be performed 100% correct, it does not mean that the whole transaction must fail [CZM10]. A user can define a completion constraint condition for a service, which expresses a kind of a situation where the service can be regarded as successfully performed taking into account the task of the transaction and the role of the service in it. The situation is expressed by writing a truth-value statement. The statement, which is the actual completion constraint condition, will be tested if the service fails [CZM10]. If the completion constraint condition will return the value true, the particular service can be ignored [CZM10]. This means that the fail of the particular service of the transaction does not harm the task of the whole transaction.

A completion constraint condition of a service is written using an expression of XPath 1.0 [CD99] which returns a Boolean value [CZM10]. If the return value is false the service could not been ignored [CZM10]. E.g. an atomic service named Contact airlines can have a completion constraint condition $sucConnection >= 1 [CZM10]. It means that the service has performed successfully when it has been able to contact at least one airline company. In that case it returns the value true. The service can try to contact several airline companies but in order to return true, one succeeded contact is enough. If it cannot contact any airline company then the service has failed and the completion constraint condition will be false.

For a compensation service there is another kind of constraint condition: an *ignorable compensation constraint condition* (ICCC). It is also a truth-value statement, which will be true if a compensation service in the backward recovery situation can be ignored [CZM10]. If the semantics or the relationships of the compensation service causes that the compensation service must be performed in the backward recovery situation, the ignorable compensation constraint condition will get the value false [CZM10]. Many ignorable compensation constraint conditions can be defined for a compensation service. In that case all of them must be true before the compensation service can be left not performed [CZM10]. So the usage of ignorable compensation constraint conditions makes it possible to leave some successfully performed services without compensation in the backward recovery situation.

An expression of XPath 1.0, which returns a Boolean value is used to write an ignorable compensation constraint condition [CZM10]. E.g., a compensation service (of the service SendChargenote) SendChargeinfo can have an ignorable compensation constraint condition $noteHtPrice=$htPrice [CZM10]. It means that if the price of the suggested hotel is same than the price of another hotel which a customer wants to have, instead the ignorable compensation constraint condition of the compensation service SendChargeinfo will be true, because there is no need to at first run the compensation service and after that the service SendChargenote with the same price. So in this case the compensation service can be ignored.

4.3 Logic Rule

A constraint rule called logic rule is defined, its usage is explained and its implementation is described. Possibility to combine constraint conditions and logic rules and use them together is explained. The management of the constraint rules is described briefly in this subsection.

It is not always sensible or effective to run the compensation services in the opposite order of the executed services [CZM10], which need to be compensated in the backward recovery situation. E.g., delivery of goods and receiving payment services of the online shopping transaction are usually performed simultaneously. Although in the backward situation, the compensation services are better to be run so that the compensation of receiving payment will be run first, because the delivery company wants to charge before they want to take care of returning of the goods [CZM10]. *Logic rules* will be used to determine a running order and a schedule of the compensation services in the backward recovery situation [CZM10]. Normally, every service has a compensation service which will clean up the effects of the executed service partly or totally [CZM10]. A logic rule of a composite service will be created during the workflow of creating the composite service [CZM10].

Logic rules can be written using BPEL [CZM10], which is an XML-based language used to determine the interaction of the web services. There can be a label <compensationHandler> [CUR03] in the BPEL labels <scope> and <process>. In the label <compensationHandler> there can be the labels <sequence> and <flow> which are used to describe the logic rules. Earlier mentioned completion constraint condition and ignorable compensation constraint condition which are written using XPath 1.0 can be

used in BPEL to cause conditional occurrence [CZM10]. So constraint conditions and logic rules can be used together.

As a scope executes a certain sub task of the eBusiness transaction and the scope has a coordinator to manage its participants, a coordinator is also an owner of a constraint rule. It means that the coordinator is aware of the constraint rule and the duty of the coordinator is to maintain and evaluate it [CZM10]. Constraint rules do not affect to the sub coordinators [CZM10] or to the services of the scopes of sub coordinators. A coordinator having several scopes which constraint rules affecting to the services of other scopes of the same coordinator is an exception [CZM10].

4.4 Constraint Rules-Based Recovery Mechanism

A Constraint rules-based recovery mechanism is presented and explained. It makes use of four techniques: an atomic service retrying, a minimum range recovery, a synchronized compensation and a customer interaction. These are introduced and a phase when they are used in the Constraint rules-based recovery mechanism is depicted and described in this subsection.

The idea of the C*onstraint rules-based recovery mechanism* is that when a failure or an exception in the execution of a scope happens, it tries to recover it by doing forward recovery and in case it was not successful, it tries to recover by doing a backward recovery [CZM10]. The process of the Constraint rules-based recovery mechanism [CZM10] is depicted in Figure 4B. When an exception occurs in a service of a scope, a forward recovery will be started and the value of a completion constraint condition of the service is evaluated. If the value of the completion constraint condition is true, the execution of the scope can continue with the consecutive services. If the value of the completion constraint condition of the service is false then it is checked only in case the exception happened in an atomic service. If the atomic service caused the exception then the coordinator of the scope tries to run that service again.

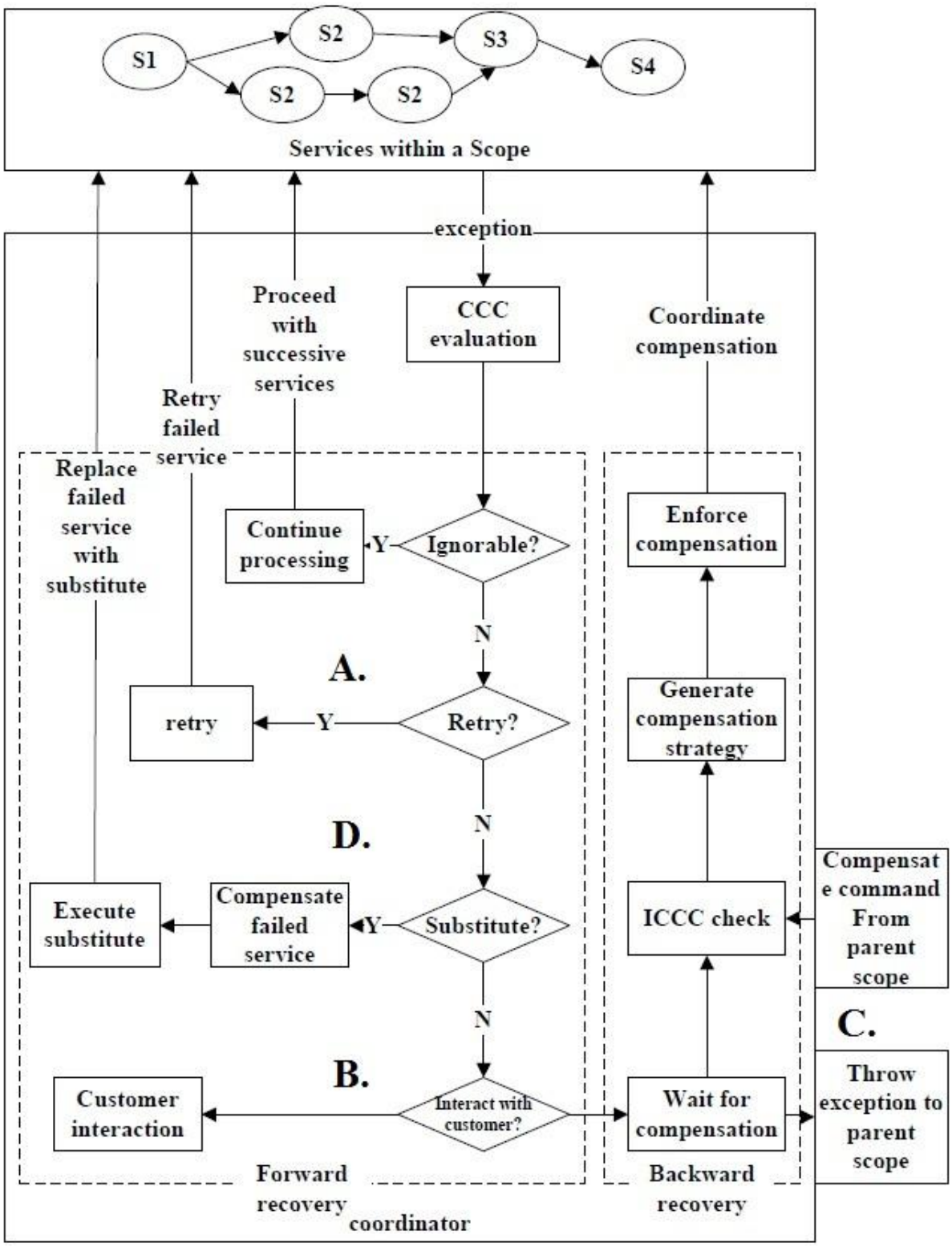

Figure 4B. A process of the recovery using a Constraint rules-based recovery mechanism. Four key techniques used in the recovery are A. an atomic service retrying, B. a minimum range recovery, C. a synchronized compensation and D. a customer interaction.

If the exception happened in a composite service of the scope then the provider of the failed service will negotiate with a customer. The negotiation is described in Figure 4C [CZM10]. The provider will suggest a substitute service or ask permission to cancel the service. If the customer chooses one of these options the constraint conditions will be changed [CZM10]. If the customer does not want to abandon the service, a compensation constraint condition will be evaluated to know if the substitution will bring on the compensation of the entire scope. If the substitution will not bring on the compensation of the entire scope, the compensation of the failed service is done by activating a substitution service. So the failed service is replaced with a substitution service.

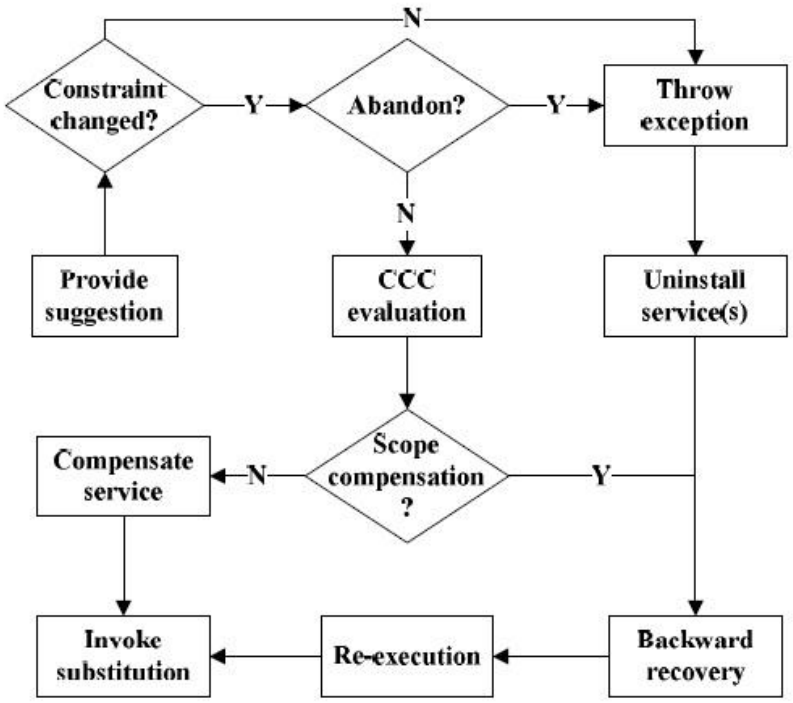

Figure 4C. An interaction between a customer and a provider of the failed service during the execution of the Constraint rules-based recovery mechanism [CZM10].

If the customer does not want to use a substitute and wants to cancel the failed service an UnatomicException will be thrown by the coordinator to the outer scope and the consecutive services of the failed service will be uninstalled so that they will not be activated [CZM10]. Same thing happens if the customer does not accept either option: the substitution or cancellation. In the two previous cases where the UnatomicException was thrown, the recovery process will stop and wait for the compensation command from the outer scope. After the outer scope has given a compensate command or in case of that

the substitution causes the compensation of the entire scope, a backward recovery will be started. An ignorable compensation constraint condition will be evaluated. If there are services that are linked to the ignorable compensation constraint condition but they are not activated yet an optimistic evaluation strategy will be used [CZM10]. The *optimistic evaluation strategy* means that the linked inactivated services are assumed to be ignorable and the ignorable compensation constraint condition will get the value true [CZM10]. It causes that there are fewer services to compensate [CZM10]. A compensation strategy will be generated, carried out and coordinated. When the backward recovery is completed the compensated services need to be run again and the substitution of the failed service will be executed as well.

The Constraint rules-based recovery mechanism makes use of four techniques: an atomic service retrying, a minimum range recovery, a synchronized compensation and a customer interaction [CZM10]. Those techniques are marked in the recovery process in Figure 4B using letters A, B, C, and D. *An atomic service retrying* (A.) means that only the atomic services (not composite ones) of the scope will be retried if they fail [CZM10]. This is because if there is an exception in the composite service all the services of that scope will be retried and the composite service must remain failed before a coordinator of the scope will throw an “UnatomicException” to the parent scope [CZM10]. After that, the recovery process will know that the composite service should be retried but there is no need to retry as all its services have already been retried. The atomic service retrying prevents the unnecessary retries of the services.

In a *minimum range recovery* (B.) a coordinator of the scope does all possible actions to forward recover a failed service within the scope [CZM10], e.g., by substituting or cancelling the failed service if the customer allows. If the coordinator will not succeed with the forward recovery, the scope has failed and an UnatomicException is thrown to the outer scope. Then the range of the influence of the failed service will be enlarged. The minimum range recovery takes care that the failed service affects only the needed number of the scopes and causes as little side effects as possible [CZM10].

A *customer interaction* (D.) is the negotiation between a customer and the provider of the failed service if the failed service can be replaced with another, cancelled or neither of them meaning that the customer does not want to compromise. The provider will suggest

to the customer possible substitutes of the failed service and will give information about them [CZM10]. Based on the customer's decision the recovery process will continue.

A *synchronized compensation* (C.) means that after a coordinator has thrown an UnatomicException as a sign of the failed forward recovery of the failed service a recovery process must wait for the compensation command of a parent scope [CZM10]. The synchronization between the scope, the parent scope and other scopes is essential in order to fulfill the logic rules of the parent scope [CZM10].

5 Analysis

The expectations for a service composition and recovery model are listed and the corresponding features of the two new models are written next to them. Those new models: the DeltaGrid service composition and recovery model and the Constraint rules-based recovery mechanism and two classic models: the Saga and the BTM are compared with each other. The explanations of the expectations are given and how the models answer to the expectations is analysed in this section.

The expectations set for a new service composition model and its recovery model are listed in Table 6. There are also the features of the DeltaGrid service composition and recovery model and the features of the Constraint rules-based recovery mechanism, which answer to those expectations. They are grouped by the aspects of the execution requirements of an eBusiness transaction. A comparison between the classic and new models will be done. The classic models are the Saga and the BTM and the new models are the DeltaGrid service composition and recovery model (DGM) and the Constraint rules-based recovery mechanism (CM).

An aspect of execution requirements of an eBusiness transaction [HA02]	DeltaGrid service composition and recovery model [UX09]	Constraint rules-based recovery mechanism [CZM10]	Expectations for a service composition and recovery model
granularity, cohesion	A **hierarchical** structure of an eBusiness transaction can be expressed as a composite group which consists of atomic groups. The functionality of a service is defined as an operation of the atomic group. A composite group can be composed of other composite groups -> **nested structure**	A **hierarchical** structure of an eBusiness transaction can be described as a scope. The scope defines a composite service. An atomic service has a provider and a composite service consist of atomic services. Participant of a scope can be a sub coordinator-> **nested structure**	**Granularity** levels from process to a service allow a **flexible hierarchical** composition structure. A transaction has **relaxed atomicity**.

coupling	The **delta schedule** is used to analyse data dependencies among concurrently executing processes when process failure occurs. If the primary operation is **dismissible** the atomic group does not need to have a compensation plan.	Relationships between atomic services and composite services can be expressed with **constraint rules** used in the recovery process. **ICCC** is used to decide if compensation is ignorable and what the execution order and the schedule of compensation services (**logic rules**) are in the backward recovery.	Transactions have a limited number of data and control **dependencies** which can be taken into account in the backward recovery.
reversibility	An atomic group has essential compensation and **contingency plans** for a primary operation of it. The need of the compensation and the **contingency plan** depends on the post-commit recoverability of the primary operation and the value of the **criticality attribute** of the atomic group. If the primary operation is compensatable the atomic group must have a compensation. If the atomic group is critical it must have the contingency. Using compensation, **contingency** and DE-rollback at the atomic and the composite groups an execution failure of the process can be automatically recovered at any composition level **maximizing the potential forward recovery.**	In the forward recovery a **CCC** makes possible to check if a service have completed successfully. The forward recovery uses **retry** (if an atomic service caused the exception: *An atomic service retrying*), (if the exception happened in a composite service negotiation: *customer interaction*) **substitute** or **permission to cancel** *minimum range recovery*= tries to forward recover within the scope. If an execution of the scope fails a **forward** recovery is tried at **first**.	A service has a mechanism for compensation and a **contingency** plan. A forward recovery is **maximized**.

reliability	A mechanism is picked from “A service has a mechanism for compensation and a contingency plan.” on the previous row.	A mechanism is picked from “A service has a mechanism for compensation and a contingency plan.” on the previous row.	A mechanism for a compensation of each **sub-transaction** is needed.
concurrency	A delta schedule is used to analyze data dependencies of **concurrently** running processes. The delta schedule supports **DE-rollback** which restores the results of the execution of a service as they were even if the execution has already terminated**.**	**Locking** is **not used** and that is why traditional rolling back cannot be used in the recovery process. Compensation is used instead of rolling back.	**Isolation** is **relaxed** which allows that data elements are not locked during the execution of the transaction.
recoverability	**Operation** is an activation of a DEGS service which is an autonomous entity that takes care of its local correctness using a local **compensation** transaction. It never terminates in the failed state. The DEGS produces and sends **deltas** to a **PHCS** which maintains the execution context of every running process in the system and creates a log file called a **delta schedule**. The operation is an ACID DEGS or a multilevel DEGS. A multi-level DEGS has a ***pre-commit recoverability*** **mechanism** (DE-rollback or a service reset function) which cleans the	In the **backward recovery** situation **logic rules** defines what is the execution order and the schedule of compensation services and **ICCC** defines if a compensation is ignorable, based on the semantics of the compensation service and its relationships to other services. If there are services that are linked to the ICCC but they are not activated yet an **optimistic evaluation strategy** will be used in a backward recovery. The synchronization between the scope, the parent scope	A transaction has **relaxed consistency**. There have to be logging, save points and context security mechanisms available so that a transaction reaches a **consistent state** if a service fails. A **recovery mechanism** is described in detailed.

	consequences of the tried pre-commit-compensation. It increases the level of the **consistency** of the multilevel DEGS operation. A ***post-commit recoverability*** mechanism of a DEGS operation (options: **reversible, compensatable**) is needed to semantically undo a successfully terminated operation. The backward recovery of the atomic group (options:**compensation, DE-rollback** and **service-reset**) always terminates in a consistent state. **Shallow and deep compensation** are used in the backward recovery of a composite group which never terminates in an inconsistent state. In the backward recovery the fault of the operation will be passed to the enclosing atomic group and the fault of the atomic group will be **passed to** the **enclosing** composite group recursively until the whole process has backward recovered.	and other scopes is essential in order that the logic rules of the parent scope will be fulfilled. In a **synchronized compensation** after a coordinator has thrown an UnatomicException a recovery process must wait the compensation command of a parent scope.	
reusability	Atomicity types are **not used**.	Atomicity types are **not used**.	**Unaccustomed** classifying of **atomicity** is used.

Table 6. The expectations set for a new service composition model and its recovery model. The features of the DeltaGrid service composition and recovery model and the features the Constraint rules-based recovery mechanism bearing to the expectations. They are grouped by the aspects of the execution requirements of an eBusiness transaction.

An explanation of the expectations for a service composition and recovery model are written as **bold**. The four service composition and recovery models, the classic models the Saga and the BTM and the new models the DGM and the CM, are evaluated after each explanation with bold text.

A recovery model is founded on a service composition model which should be hierarchical and well defined starting from a service up to the expression of the entire eBusiness process. It should allow a nested composition structure which has a needed number of nesting levels to describe an entire process using services. A Saga is composed of sequential transactions and has only two nesting levels: a Saga consists of atomic simple transactions. A long running transaction (LRT) of the BTM is better because it is composed of the atomic transactions and it uses an open nested transaction model allowing the needed number of nesting levels. This way an eBusiness process can be described using nested LRTs. An operation of the DGM and an atomic service of the CM are similar to the simple transaction of the BTM. The service composition structure of DGM and CM are similar to the BTM because the hierarchical service structure of a scope of the CM is similar to the composite group of the DGM that is similar to a LRT of the BTM. All structures are composed of smaller atomic parts, allow open nested transaction structure and an eBusiness process can be expressed using them. In case of the composite group of the DGM the parts are other composite groups and atomic groups. Also the atomic group includes service activation and it is a building block of composite group. In case of the scope of the CM, the parts are other scopes, meaning composite services, and atomic services, which are the building blocks of a composite service. In case of the LRT of the BMT the parts are other LRTs and atomic transactions. So the service composition model of the DGM and the CM are as good as the BTM.

A transaction has to relax atomicity. A Saga relaxes atomicity by allowing access to the shared recourses but if the transaction of the Saga fails all the transactions of the Saga must be compensated which is not practical. That is why the Saga does not properly relax atomicity. The LRT of the BTM relaxes atomicity better, because its participants can individually decide either to commit or to rollback an atomic transaction of the long running transaction. So the LRT relaxes atomicity using open nested transactions. The DGM relaxes the atomicity defining a criticality attribute for an atomic group. The CM relaxes

atomicity defining a CCC for a service. So the DGM and CM relax the atomicity using the same principle and define a rule for checking it which the BTM not have.

The data and control dependencies are needed in order to take into account the backward recovery of a transaction. A Saga cannot restore the data taking into account the data dependencies to other transactions but a LRT can do so if the backward recovery is performed together with compensation and definition of the business logic of the long running transaction. The LRT does not describe how the business logic can be defined, but in CM the business logic is defined using logic rules. In the backward recovery situation they are used to define the execution order and the schedule of compensation services. The logic rule of a composite service is created along with the workflow in a phase of the service composition. In addition, an ignorable compensation constraint condition (ICCC) will be checked if a compensation of the single service is ignorable. In DGM the logic of the eBusiness transaction is collected in a delta schedule during the execution of the service of the process. The delta schedule is a part of DeltaGrid environment and it is used to analyse data dependencies among concurrently executing processes when a backward recovery is needed. ICCC matches up to dismissible, which is the post-commit recoverability option of the primary operation in DGM. Both models DGM and CM take into account the business logic automatically in the backward recovery and offer a mechanism to ignore an unnecessary compensation. So in that way, the two models are equal but they have been improved compared to the LRT of the BTM.

A service should have a mechanism for compensation and a contingency plan and a forward recovery should be maximized. Every simple transaction of a Saga has a compensation transaction. The Saga has a pure forward recovery as a contingency plan if the save-point command is run automatically at the beginning of each transaction. If the pure forward recovery is not possible backward-forward recovery will be performed: the compensation transactions are executed to reach the save-point and rerunning the transactions will be started. If the compensation transaction or the pure forward recovery fails there are three options: a retry, an alternate transaction, a manual intervention (does not hold data resources). A LRT has compensating actions to cancel the effects of the failure [PAP03]. If a system failure occurs during the execution of the LRT the transaction performs a forward recovery by returning a consistent state and continuing catering for the occurred failure. The BTM does not represent the forward recovery more detailed. So the

Saga gives better description of the forward recovery than the LRT because it offers two different ways to do so and also three ways to handle to failed forward recovery.

Each atomic service of the CM has a compensation but if an execution of the scope fails a forward recovery is tried before backward recovery. When a service fails in the CM, a completion constraint condition (CCC) is checked. If CCC is not satisfied it means that a service has not completed successfully and a forward recovery will be started. If the failed service is atomic an atomic service retrying is performed meaning the coordinator retries to run the service. In case of a composite service the provider of the failed service will negotiate with a customer (customer interaction) asking for a permission to substitute or to cancel. So a minimum range recovery is used. If the forward recovery fails an exception is thrown to the parent scope. The forward recovery of CM has the same recovery options as the failed compensation or the failed pure forward recovery of the Saga: a retry, a substitution (=alternate transaction) and a customer interaction, which demands an action of the customer as the manual intervention of the Saga demands an action of the programmer. So CM does the forward recovery with the same actions as the Saga handling the recovery of the failed compensation or the failed pure forward recovery. That can be called maximizing the forward recovery and it is an improvement compared to the Saga. The CM has also added the customer interaction and CCC checking, which the Saga does not have.

An atomic group of DGM has on needs basis compensation and contingency plans for a primary operation of it. The need of the compensation and the contingency plan depends on the post-commit recoverability of the primary operation and the value of the criticality attribute of the atomic group. If the primary operation is compensatable the atomic group must have a compensation whereas a reversible or a dismissible primary operation does not need it. If the atomic group is critical, it must have the contingency. When the primary operation has failed and compensated, the contingency is performed in case the atomic group is critical. Otherwise, the contingency is not run. If the contingency fails pre-commit compensation options: compensation, DE-rollback and service reset, which demands a special program or a human agent are tried. So DGM does not use a retry as a forward recovery option. Instead of them the DGM guarantees the consistent state of the atomic group in the failure of the forward recovery by using pre-commit compensation options that make use of the DE-rollback of the DeltaGrid environment and an automatic rollback and service reset of the service provider. So the forward recovery of the DGM is closer to the

backward-forward recovery of the Saga than the CM, but the DGM's version is more effective than the Saga because the backward recovery also has automated options. Using compensation, contingency and DE-rollback at the atomic and the composite groups an execution failure of a process can be automatically recovered in the DGM at any composition level. That is maximizing the potential forward recovery [UX09].

A mechanism for a compensation of each sub-transaction is needed. All the models have a mechanism for a compensation of each sub-transaction. A simple transaction of the Saga has a compensation transaction. A LRT has compensating actions but they are not described more detailed. An atomic service of the CM has a compensation and an atomic group of the DGM has on needs basis compensation plan for a primary operation. The Saga, the CM and DGM defines a compensation and specifies the usage of it. The BTM does not.

A transaction has a relaxed isolation. A Saga relaxes isolation by letting a transaction of the Saga commit without taking into account if the other transactions of the same Saga are committed, but it may cause an inconsistency problem which the Saga cannot solve. A LRT of the BTM relaxes isolation the same way but it is able to handle the possible inconsistency problem. In the CM, isolation is relaxed by not allowing locking. For this reason a traditional rolling back cannot be used in the recovery process of CM. Compensation is used instead of rollback. As well in the DGM, the locking of recourses is not used, which relax isolation. The DGM differs from the Saga, the BTM and the CM so that a service can be rolled back. That action is called DE-rollback and it is done using the delta schedule, which restores the results of the execution of a service even if the service has been terminated.

A transaction has relaxed consistency. There have to be logging, save points and context security mechanisms available so that a transaction reaches a consistent state if a service fails. A recovery mechanism is described in detailed. A Saga relaxes consistency alternatively with a pure forward recovery or by defining a compensation transaction for each transaction of the Saga. The pure forward recovery requires that the save-point command is in use. In the latter case, the Saga cannot solve the inconsistency problem caused by simultaneous relaxation of isolation although the Saga execution component uses a log [GS87]. A LRT of the BTM relaxes consistency by running a forward recovery and running a backward recovery together with compensation

and definition of the business logic of the long running transaction. The backward recovery is not described more detailed. So the Saga describes the recovery more detailed than the BTM.

The backward recovery is not simple to implement because each operation of the service would need to lock the resources and have a pre-defined compensation [CO08]. That is a reason a forward recovery is preferred for the LRTs [CO08]. Both keep a transaction in a consistent state. The LRT of the BTM takes care of the consistency better than the Saga.

The CM relaxes consistency by defining a CCC for a service, which indicates if the service has been performed successfully taking into account the task of the scope and the role of the service in it. When a service fails, a CCC is checked. If CCC is not satisfied a forward recovery will be started. If the forward recovery did not succeed a coordinator of the scope will throw an UnatomicException for the parent scope. A synchronized compensation is used, which means that a backward recovery process must wait the compensation command of a parent scope. The synchronization between the scope, the parent scope and other scopes is essential in order to fulfil the logic rules of the parent scope. After the outer scope has given a compensate command a backward recovery will be started. An ICCC is evaluated. If there are services that are linked to the ICCC but they are not activated yet, an optimistic evaluation strategy will be used. A compensation strategy will be generated using the optimistic evaluation strategy and logic rules. Logic rules define the execution order and the schedule of compensation services. Then the backward recovery is carried out and the process reaches a consistent state. The CM describes the backward recovery more in detail than the BTM. It also defines new strategy and helpful rules for a process to reach a consistent state. The CCC and ICCC describe the relationships of the services of the process. The logic rules and the optimistic evaluation strategy take into account the business logic in the backward recovery. With the optimistic evaluation strategy there are fewer services to compensate.

The DGM relaxes consistency with a multi-level DEGS operation and a pre-commit recoverability mechanism helps to reach a consistent state by eliminating the state failed. An operation of the DGM is an activation of a DEGS service which is an ACID DEGS operation or a multilevel DEGS operation. The ACID DEGS operation has a transaction which can automatically do rollback by underlying data base if the operation fails. So ACID DEGS operation never terminates in the failed state. The multilevel DEGS operation

consists of the ACID DEGS operations which can commit unilaterally. If one of them fails, a local compensating transaction will be run. After that, the operation will reach a consistent state. The DEGS operation produces and sends deltas to a PHCS which maintains the execution context of every running process in the system and creates a time-ordered schedule of data changes, a log file called a delta schedule. It is a foundation for DE-rollback.

A multi-level DEGS has a *pre-commit recoverability* mechanism (options: DE-rollback and a service reset function), which cleans the consequences of the tried pre-commit-compensation. It increases the level of the consistency of the multilevel DEGS operation. A post-commit recoverability mechanism of the DEGS operation (options: reversible, compensatable) is needed to semantically undo a successfully terminated operation. The operation is a part of an atomic or a composite group.

The backward recovery options of the atomic group are compensation, DE-rollback and service-reset. They guarantee that the atomic group always backward recovers to a consistent state. Shallow and deep compensation are used in the backward recovery of a composite group. The backward recovery of the composite group never terminates in an inconsistent state. In the backward recovery the fault of the operation will be passed to the enclosing atomic group and the fault of the atomic group will be passed to the enclosing composite group recursively until the whole process has backward recovered and reached a consistent state. The DGM describes the backward recovery more in detail than the CM and offers also the algorithms of the important mechanisms e.g. for the atomic group to post-commit recover and for the composite group to deep compensate. It defines many new things compared to the CM. The DGM defines a new action called DE-rollback, two new mechanisms: a pre-commit recoverability and post-commit recoverability and it also extends the concepts of a shallow compensation and a deep compensation. The use of them guarantees that a process recovers always in a consistent state which the Saga and the BTM could not proof.

Unaccustomed classifying of atomicity is used. The BTM has defined atomicity types for a LRT, which corresponds to an unaccustomed classifying of the atomicity. The DGM and the CM do not have or use them. It could be good to map the atomicity types, which are higher level transaction requirements onto combinations of lower level, basic

transactions models [CO08]. Thus, the DGM and the CM are missing a feature that the BTM has and maybe DGM and CM should include it as well.

6 Conclusions

The nature of the eBusiness collaboration sets requirements for the long running transactions. E.g., the ACID-properties must take a relaxed form when the long running eBusiness transactions are managed. Many techniques have been developed to take care of the execution of the long running business transactions. As an example of the classic service composition and recovery model the classic Saga and a business transaction model (BTM) of the business transaction framework (BTF) were introduced.

The expectations for a new service composition and recovery model were set. They were grouped by the aspects of the execution requirements of an eBusiness transaction. As an example of the new service composition and recovery model the DeltaGrid service composition and recovery model (DGM) and the Constraint rules-based recovery mechanism (CM) were introduced. The explanations of the expectations were given. The four models were compared to each other and it was analysed how the models answer to the explanations of the expectations.The result of the analyses answers to the research question of this paper: how to secure a long running eBusiness transaction to a consistent state through recovery during an eBusiness transaction.

A service composition model of the new models is as good as the BTM. Both new models have improved the management of the relaxed atomicity compared to the classic models by defining a rule (a completion constraint condition (CCC) of the CM and the criticality attribute of the DGM) for checking if a service does not need to be successful to be able to terminate the enclosing transaction successfully.

A recovery model of the new models has improved the ability to take into account the data and control dependencies in the backward recovery. The Saga could not take them into account at all. The BTM presented that the business logic should be taken into account but it did not explain how it can be done. The CM uses the optimistic evaluation strategy, which results to fewer services to be compensated. Both new models take into account the business logic automatically in the backward recovery which is a great improvement.

The new models present two different kinds of strategies to recover a failed service. The CM tries to do the forward recovery first and in case it did not succeed the backward recovery is started. That increases the flexibility and the efficiency [CZM10] compared to

the Saga or the BTF. The CM uses a retry as a forward recovery option which the DGM does not have. The DGM does the backward recovery first using compensation or DE-rollback and after that starts the forward recovery. Both new models offer a mechanism to ignore an unnecessary compensation: an ignorable compensation constraint condition (ICCC) of the CM and a post-commit recovery option dismissible of the DGM. Both new models also maximize the potential forward recovery compared to the classic models.

The CM does not describe how to clean up the effects of the failed operation before using retry as a forward recovery option. The DGM has a service reset function for it. The CM is lacking the three pre-commit recoverability options out of four compared to the DGM: automatic rollback, DE-rollback and service reset. On the other hand, if the process is not performed in the DeltaGrid environment then the automatic rollback and the DE-rollback are not available. The CM does not describe what happens if the ICCC specifies that the compensation must be performed and the outer scope has given a compensate command, but the compensation of the service is missing. The DGM handles the situation by running DE-rollbackOrServiceReset algorithm.

The DGM describes the backward recovery more in detail compared to the CM and offers also the algorithms of the important mechanisms. It defines characteristics that the CM does not have: a DE-rollback, mechanisms for a pre-commit recoverability and for a post-commit recoverability and extends the concepts of a shallow compensation and a deep compensation. The use of them guarantees that an eBusiness process recovers always in a consistent state which is something the Saga, the BTM and the CM could not proof.

Neither new model uses the unaccustomed classifying of atomicity even if the BTM includes the unaccustomed classifying of atomicity. A future direction is to study how to map the higher level transaction requirements e.g. atomicity types onto transaction model of these new models.

References

AL07 Alves A. et. al.: Web Services Business Process Execution Language Ver 2.0 OASIS Standard April 2007. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel Checked 19.2.2012

CAA05 Cabrera F. et. al.: Web Services Atomic Transaction (WS-AtomicTransaction) Ver. 1.0. August 2005. http://www.ibm.com/developerworks/library/specification/ws-tx/#coor Checked on 18.2.2012

CAB05 Cabrera F. et. al.: Web Services Business Activity Framework (WS-BusinessActivity) Ver. 1.0. August 2005. http://www.ibm.com/developerworks/library/specification/ws-tx/#coor Checked on 18.2.2012

CAC05 Cabrera F. et. al.: Web Services Coordination (WS-Coordination) Ver. 1.0. August 2005. http://www.ibm.com/developerworks/library/specification/ws-tx/#coor Checked on 17.2.2012

CB97 Crawley C. J., Bukhres O.: Failure Handling in CORBAflow: A CORBA-based Transactional Workflow Architecture. Database Systems for Advanced Applications ’97, Proceedings of the Fifth International Conference on Database Systems For Advanced Applications Melbourne Australia April 1-4, 1997.

CD99 Clark J., DeRose S.: XML Path language (XPath) Version 1.0, W3C Recommendation, 1999.

CO08 COMPAS: State-of-the-art in the field of compliance languages. EU Project COMPAS Compliance-driven Models, Languages, and Architectures for Services, Deliverable D2.1, version 1.0. 2008-07-31. http://ec.europa.eu/information_society/apps/projects/logos/5/215175/080/deliverables/D2.1_State-of-the-art-for-compliance-languages.pdf Checked on 11.2.2012.

CUR03 Curbera F. et. al.: Business Process Execution Language for Web Services (BPEL4WS) 1.1 May 2003. http://www.ibm.com/developerworks/library/specification/ws-bpel/ Checked on 10.2.2012.

CZM10 Cao J., Zhang B., Mao B., Liu B.: Constraint Rules-based Recovery for Business Transaction. Grid and Cooperative Computing(GCC), 2010 9th International Conference on 1-5 Nov. 2010. Pages: 282 – 289 DOI: 10.1109/GCC.2010.63

GLA02 Grefen P., Ludvig H., Angelov S.: A framework for e-Services: A three-level Approach towards Process and Data management. In 02-07, C.T.R. editor University of Twente, 2002.

GS87 Garcia-Molina H., Salem K.: Sagas. ACM SIGMOD Record Volume 16, Issue 3, 1987. Pages: 249 - 259. DOI http://doi.acm.org/10.1145/38714.38742

HA02 Heuvel, W.J. van den, Artyshechev: Developing a three-dimensional transaction model for supporting atomicity spheres. Proceedings oy NetObjectDays 2002 vol 2.

HV82 Hadzilacos, Vassos: An algorithm for Minimizing Roll Back Cost. Proc ACM Symp on PODS, Los Angeles, CA, March 1982. Pages: 93-97.

LA95 Laymann F.: Supporting business transactions via partial backward recovery in workflow management. Proc. of the GI-Fachtagung fur Datenbanksysteme in Business, Technologie and Web(BTW'95), 1995.

PK06 Papazoglou M., Kratz B.: A business-aware web services transactions model. Proceedings of the 4th International Conference on Service-Oriented Computing 2006.

PAP03 Papazoglou M. P.: Web Services and Business Transactions. World Wide Web Volume 6, Number 1, March, 2003. Pages: 49 - 91 DOI: 10.1023/A:1022308532661

SH05 Singh M. P., Huhns M. N.: Service-Oriented Computing Semantics, Processes, Agents. John Wiley & Sons Ltd., 2005.

SHH05 Seunglak C., Hyukjae J., Hangkyu K., Jungsook K., Su Myeon K., Junehwa S. and Yoon-Joon L.: Maintaining Consistency Under Isolation Relaxation of Web Services Transactions. Kitsuregawa et al. (Eds.): WISE 2005, LNCS 3806. Pages: 245 – 257. 2005. © Springer-Verlag Berlin Heidelberg 2005.

UX09 Urban S. D., Xiao Y.: The DeltaGrid Service Composition and Recovery Model. International Journal of Web Services Research, vol. 6, no. 3, 2009. Pages: 35-66.

UXB09 Urban S., Xiao Y., Blake L., Dietrich S.: Monitoring data dependencies in concurrent process execution through delta-enabled grid services. International Journal of Web and Grid Services 2009 - Vol. 5, No.1. Pages: 85 – 106. DOI: 10.1504/IJWGS.2009.023870