The DeltaGrid Service Composition and Recovery Model

Yang Xiao and Susan D. Urban
Arizona State University
School of Computing and Informatics
Department of Computer Science and Engineering
Tempe, AZ 85287-8809
s.urban@asu.edu

ABSTRACT:

This research has defined an abstract execution model for establishing user-defined correctness and recovery in a service composition environment. The service composition model defines a flexible, hierarchical service composition structure, where a service is composed of atomic and/or composite groups. The model provides multi-level protection against service execution failure by using compensation and contingency at different composition granularity levels, thus maximizing the potential for forward recovery of a process when failure occurs. The recovery procedures also includes rollback as a recovery option, where incremental data changes known as deltas are extracted from service executions and externalized by streaming data changes to a Process History Capture System. Deltas can then be used to backward recover an operation through a process known as Delta-Enabled Rollback. Our work defines the semantics of the service composition model and the manner in which compensation, contingency, and DE-rollback are used together to recover process execution. We also present a case study and describe a simulation and evaluation framework for demonstrating the functionality of the recovery algorithm and for evaluating the performance of the recovery command generation process.

KEY WORDS:

service composition, compensation, contingency, delta-enabled rollback, failure recovery

1. INTRODUCTION

In a service-based architecture, a process is composed of a series of calls to distributed Web services and Grid services that collectively provide some specific functionality of interest to an application (Singh and Huhns, 2005). In a traditional, data-oriented, distributed computing environment, a distributed transaction is used to provide certain correctness guarantees about the execution of a transaction over distributed data sources. In particular, a traditional, distributed transaction provides all-or-nothing behavior by using the two-phase commit protocol to support atomicity, consistency, isolation, and durability (ACID) properties (Kifer et al., 2006). A process in a service-oriented architecture, however, is not a traditional ACID transaction due to the loosely-coupled, autonomous, and heterogeneous nature of the execution environment. When a process invokes a service, the service performs its function and then terminates, without regard for the successful termination of the global process that invoked the service. If the process fails, reliable techniques are needed to either 1) restore the process to a consistent state or 2) correct critical data values and continue running.

Techniques such as compensation and contingency have been used as a form of recovery in past work with transactional workflows (e.g., Worah and Sheth, 1997) and have also been introduced into recent languages for service composition (e.g., Lin and Chang, 2005). In the absence of a global log file, compensation provides a form of backward recovery, executing a
procedure that will “logically undo” the effects of completed and/or partially executed operations. Contingency is a form of forward recovery, providing an alternate execution path that will allow a process to continue execution. Some form of compensation may be needed, however, before the execution of contingency plans. Furthermore, nested service composition specifications can complicate the use of compensating and contingent procedures. To provide a reliable service composition mechanism, it is important to fully understand the semantics and complementary usage of compensation and contingency, as well as how they can be used together with local and global database recovery techniques and nested service composition specifications.

This research has defined an abstract execution model for establishing user-defined correctness and recovery in a service composition environment. The research has been conducted in the context of the DeltaGrid project, which focuses on building a semantically-robust execution environment for processes that execute over Grid Services (Xiao, 2006; Xiao, Urban, and Dietrich, 2006; Xiao, Urban, and Liao, 2006; Xiao and Urban, 2007a). This paper is an extended version of the work presented in (Xiao, Urban, and Liao, 2006), with a focus on the full specification of the abstract service composition and recovery model for the DeltaGrid environment.

The service composition model defines a flexible, hierarchical service composition structure, where a service is composed of atomic and/or composite groups. An atomic group is a service execution with optional compensation and contingency procedures. A composite group is composed of two or more atomic and/or composite groups and can also have optional compensation and contingency procedures. A unique aspect of the model is the provision of multi-level protection against service execution failure by using compensation and contingency at different composition granularity levels, thus maximizing the potential for forward recovery of a process when failure occurs.

Another unique aspect of the model is the support it provides for rollback as a recovery option. Distributed services in the DeltaGrid environment, referred to as Delta-Enabled Grid Services (DEGS), are extended with the capability of recording incremental data changes, known as deltas (Blake, 2005; Urban et al., 2007). Deltas are extracted from service executions and externalized by streaming data changes out of the database to a Process History Capture System (PHCS) (Xiao, Urban, and Dietrich, 2006). The PHCS merges deltas from distributed sources into a time-ordered schedule of the data changes associated with concurrently executing processes. Deltas can then be used to backward recover an operation through a process known as Delta-Enabled Rollback (DE-Rollback) (Xiao, 2006). DE-rollback can only be used, however, if certain recoverability conditions are satisfied, with the PHCS and the merged schedule of deltas providing the basis for determining the applicability of DE-rollback based on data dependencies among concurrently executing processes.

Our work defines the semantics of the service composition model and the manner in which compensation, contingency, and DE-rollback are used together to recover process execution. The results presented in this paper outline recovery algorithms in the context of single process execution, defining procedures for the application of shallow versus deep compensation, and also defining conditions for the applicability of the different recovery options. We also present a case study and describe a simulation and evaluation framework that we developed to demonstrate the functionality of the recovery algorithm and to also evaluate the performance of the recovery command generation process.

The rest of the paper is organized as follows. After outlining related work in Section 2, Section 3 gives an overview of the DeltaGrid system that provides the context and test bed for this research. Section 4 provides an overview of the service composition and recovery model. defines the compositional structure of the model and defines the semantics of each compositional element of the model. Section 5 then elaborates on the semantics of the composition model, Section 6 presents algorithms for service recovery based on the semantics of each execution entity. A case study is given in Section 6, illustrating the use of the model in the context of an
online shopping application. Section 7 describes our results with the simulation and evaluation of the recovery algorithms. The paper concludes in Section 8 with a summary and discussion of future research.

2. RELATED WORK

The traditional notion of transactions with ACID properties is too restrictive for the types of complex transactional activities that occur in distributed applications, primarily because locking resources during the entire execution period is not applicable for Long Running Transactions (LRTs) that require relaxed atomicity and isolation (Cichocki, 1998). Advanced transaction models have been proposed to better support LRTs in a distributed environment (deBy et al., 1998; Elmagarmid, 1992), including the Nested Transaction Model, the Open Nested Transaction Model, Sagas, the Multi-level Transaction Model and the Flexible Transaction Model. These advanced transaction models relax the ACID properties of traditional transaction models to better support LRTs and to provide a theoretical basis for further study of complex distributed transaction issues, such as failure atomicity, consistency, and concurrency control. These models have primarily been studied from a research perspective and have not adequately addressed recovery issues for transaction failure dependencies in loosely-coupled distributed applications.

Transactional workflows contain the coordinated execution of multiple related tasks that support access to heterogeneous, autonomous, and distributed data through the use of selected transactional properties (Worah and Sheth, 1997). Transactional workflows require externalizing intermediate results, while at the same time providing concurrency control, consistency guarantees, and a failure recovery mechanism for a multi-user, multi-workflow environment. Concepts such as rollback, compensation, forward recovery, and logging have been used to achieve workflow failure recovery in projects such as the ConTract Model (Wachter and Reuter, 1992), the Workflow Activity Model (Eder and Liebhart, 1995), the CREW Project (Kamath and Ramamritham, 1998), the METEOR Project (Worah and Sheth, 1997), and Units of Work (Bennett et al., 2000). These projects expose the weaknesses of using ATM techniques alone to support reliable transactional workflow execution, mainly due to the complexity of workflows. Previous work also shows the weakness of ATMs in support of the isolation, failure atomicity, timed constraints, and liveness requirements of distributed transactional workflows (Kuo et al. 2002). Similar concerns are voiced in papers addressing transactional issues for traditional workflow systems (Alonso et al. 1997; Kamath and Ramamritham 1996; Kamath and Ramamritham 1998) as well as workflow for loosely-coupled distributed sources such as Web Services (Fekete et al. 2002; Kuo et al. 2002). More comprehensive solutions are needed to meet the requirements of transactional workflows (Worah and Sheth 1997).

In the Web Services platform, WS-Coordination (2005) and WS-Transaction (2005) are two specifications that enable the transaction semantics and coordination of Web Service composition using Atomic Transactions (AT) for ACID transactions and Business Activity (BA) for long running business processes. The Web Services Transaction Framework (WSTx) (Mikalsen et al., 2002) introduces Transactional Attitudes, where service providers and clients declare their individual transaction capabilities and semantics. Web Service Composition Action (WSCA) (Tartanoglu, 2003) allows a participant to specify actions to be performed when other Web Services in the WSCA signal an exception. An agent based transaction model (Jin and Goshnlick, 2003) integrates agent technologies in coordinating Web Services to form a transaction. Tentative holding is used in (Limthanmaphon and Zhang, 2004) to achieve a tentative commit state for transactions over Web Services. Acceptable Termination States (Bhiri et al., 2005) are used to ensure user-defined failure atomicity of composite services, where application designers specify the global composition structure of a composite service and the acceptable termination states.
In contrast, our research is among the first to provide a flexible hierarchical composition structure with automatic execution failure recovery capability for service composition (Xiao, 2006). Our research maximizes the forward recovery of a process by allowing flexible specification of multi-level contingency and compensation and addressing potential failure of recovery procedures. More importantly, we capture process execution history based on incremental data changes that can be used to backward recover a failed operation/process execution, and to support application-dependent correctness for multi-process execution. This paper presents the full details of the service composition model for the process specification and automatic service failure recovery algorithms in the context of global process execution.

3. OVERVIEW OF THE DELTAGRID SYSTEM

A unique aspect of the service composition and recovery model presented in this paper is the ability to use incremental data changes extracted from service executions as a log file to support what we refer to as Delta-Enabled rollback (DE-rollback) as an additional option for recovery of a failed process. Before presenting the composition and recovery model, this section provides an overview of the DeltaGrid environment, describing Delta-Enabled Grid Services (DEGS) and the manner in which they work together with the Process History Capture System to support the use of DE-rollback.

3.1 Delta-Enabled Grid Services

A foundational component of the DeltaGrid environment is the notion of a Delta-Enabled Grid Service (DEGS) (Blake, 2005; Urban et al., 2007). A DEGS is a Grid Service that has been enhanced with an interface that provides access to the incremental data changes, or deltas, associated with service execution in the context of globally executing process. A DEGS uses an OGSA-DAI Grid Data Service (Foster, 2001; IBM, 2005) for database interaction, modifying the SQLUpdate activity feature for database access to provide the functionality necessary for capturing and returning delta values.

The database accessed by a DEGS captures deltas using capabilities provided by most commercial database systems. Our own implementation has experimented with the use of triggers as a delta capture mechanism, as well as the Oracle Streams capability (Oracle, 2005). Oracle Streams is a feature that monitors database redo logs for changes and publishes these changes to a queue to be used for replication or data sharing. Deltas captured from the source database are stored in an associated delta repository. Deltas can then be sent to the DeltaGrid event processor after the completion of an operation execution by push mode, or be requested by the DeltaGrid system by pull mode.

A DEGS uses the object delta concept originally defined in (Sundermeir et al. 1997) to create a conceptual view of relational deltas. As shown in Figure 1, each tuple of a relation can be associated with an instance of a DeltaObject. A DeltaObject has a className indicating the name of a class (i.e., relation) to which the associated object belongs, and an objectId (i.e., primary key) to uniquely identify the associated object instance. A DeltaObject can have multiple DeltaProperty objects, which correspond to the attributes of a relation. Each DeltaProperty object has a PropertyName, and one or more PropertyValue objects (i.e., delta values). A PropertyValue contains an oldValue and a newValue, representing the old attribute value and the new attribute value, respectively. Each PropertyValue is associated with a DataChange object. The DataChange object has a processId and an operationId, indicating the global process and operation that has created the PropertyValue, with a timestamp to record the actual time of change. Deltas are extracted from the
delta repository and communicated to the delta event processor in an XML format that captures the object structure shown in Figure 1.

![Figure 1. The Delta Structure of a DEGS](image)

### 3.2 Delta-Enabled RollBack

Deltas generated by a DEGS are forwarded as a stream of information to the DeltaGrid event processor and then communicated to the Process History Capture System (PHCS). The PHCS maintains the execution context of each active process in the system. As an extension of the execution context, the PHCS merges the deltas received from each DEGS in the environment to create a time-ordered schedule of data changes for concurrently executing processes. The work in (Xiao, 2006; Xiao, Urban, and Dietrich, 2006; Urban et al., 2007) elaborates on the structure and formation of this merged schedule of data changes.

The delta schedule forms a global log file that is used for two primary purposes within the DeltaGrid system. One use of the schedule is to analyze data dependencies among concurrently executing processes when process failure occurs. In the DeltaGrid system, processes cannot enforce serializability as a correctness criterion since a process is not an ACID transaction. Processes are long running execution entities and do not lock data during the entire execution period because of the autonomy of individual Grid Services. Due to relaxed isolation, multiple processes might have interleaved access to the same data object. When one process fails, the schedule of data changes can be used to identify data dependencies and determine how the failure and recovery of one process potentially affects other executing processes that have accessed the same data (Xiao, 2006; Xiao and Urban, 2007a).

Another use of the schedule is to support *Delta-Enabled rollback (DE-rollback)*. DE-rollback is the action of reversing the data changes that have been introduced by a service execution to their before-execution images. DE-rollback can be used to reverse the results of a service execution even after the execution has terminated.

Figure 2 illustrates the execution of two processes, $p_1$ and $p_2$. The process $p_1$ is composed of two service invocations, indicated by $op_{11}$ and $op_{12}$. The process $p_2$ is composed of service executions $op_{21}$ and $op_{22}$. Each process accesses $X$ and $Y$, with the schedule of data changes shown as $x_1$, followed by $x_2$ and $x_3$, as well as $y_1$ followed by $y_2$. As shown in Figure 2, if DE-rollback is invoked on $op_{22}$, the object delta created by $op_{22}$ ($y_2$) will be removed, and the value of $Y$ will be
restored to the value y1. Likewise, DE-rollback can be applied to op21 to reverse the value of X from x3 to x2.

Since DE-rollback restores object values based on the order of modification, DE-rollback can only be invoked under semantic conditions that conform to the traditional notion of recoverability. A recoverable schedule requires that, at the time when each transaction ti commits, every other transaction tj that wrote values read by ti has already committed (Kifer et al., 2006). Thus a recoverable schedule does not allow dirty writes to occur. In a recoverable schedule, a transaction ti cannot be rolled back if another transaction tj reads or writes data items that have been written by ti, since this may cause lost updates. When interleaved access to the same data item disables the applicability of DE-rollback on an operation, compensation can be used to semantically undo the effect of the operation.

The following section elaborates on the manner in which DE-rollback, compensation, and contingency are used together to support process recovery.

## 4. SERVICE COMPOSITION AND RECOVERY MODEL

In the DeltaGrid environment, a process is hierarchically composed of different types of execution entities. Table 1 shows seven execution entities defined in the service composition model. Figure 3 uses a UML class diagram to graphically illustrate the composition relationship among these execution entities. A process is a top-level execution entity that contains other execution entities. A process is denoted as p, where p represents a process and the subscript i represents a unique identifier of the process. An Operation represents a service invocation, denoted as opi, such that op is an operation, i identifies the enclosing process pi, and j represents the unique identifier of the operation within pi. Compensation (denoted as copi) is an operation intended for backward recovery, while contingency (denoted as topi) is an operation used for forward recovery.

<table>
<thead>
<tr>
<th>Entity Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>A DEGS service invocation, denoted as opi</td>
</tr>
<tr>
<td>Compensation</td>
<td>An operation that is used to undo the effect of a committed operation, denoted as copi</td>
</tr>
<tr>
<td>Contingency</td>
<td>An operation that is used as an alternative of a failed operation (opi), denoted as topi</td>
</tr>
<tr>
<td>Atomic Group</td>
<td>An execution entity that is composed of a primary operation (opi), an optional compensation (copi), and an optional contingency operation (topi), denoted as agi = &lt;opi</td>
</tr>
<tr>
<td>Composite Group</td>
<td>An execution entity that is composed of multiple atomic groups or other composite groups. A composite group can also have an optional compensation and an optional contingency, denoted as cgi = &lt;agi</td>
</tr>
<tr>
<td>Process</td>
<td>A top level composite group, denoted as p</td>
</tr>
<tr>
<td>DE-rollback</td>
<td>An action of undoing the effect of an operation by reversing the data values that have been changed by the operation to their before images, denoted as dopi</td>
</tr>
</tbody>
</table>
An atomic group and a composite group are logical execution units that enable the specification of processes with complex control structure, facilitating service execution failure recovery by adding scopes within the context of a process execution. An atomic group contains an operation, an optional compensation, and an optional contingency. A composite group may contain multiple atomic groups, and/or multiple composite groups that execute sequentially or in parallel. A composite group can have its own compensation and contingency as optional elements. A process is essentially a top level composite group.

An atomic group is denoted as $a_{ij}$, while a composite group is denoted as $c_{ij}$. The subscripts in the atomic group and composite group notation indicate the nesting levels of an atomic group or composite group within the context of a process. For example, a process $p_i$ is a top-level composite group denoted as $c_{i1}$. Assume $c_{i1}$ contains two composite groups and an atomic group. The enclosed composite groups are denoted as $c_{i11}$ and $c_{i12}$, and the atomic group is denoted as $a_{i13}$. Assume $c_{i11}$ contains two atomic groups. These atomic groups are denoted as $a_{i111}$ and $a_{i112}$, respectively.

The only execution entity not shown in Figure 3 is the DE-rollback entity. DE-rollback is a system-initiated operation that uses the deltas of the PHCS to reverse the execution of a completed operation.

Figure 4 shows an abstract view of a sample process definition based on the DeltaGrid service composition structure. This sample process will be used throughout the rest of the paper to demonstrate different operation failure and recovery scenarios. A process $p_i$ is a top-level composite group $c_{i1}$. The process $p_i$ is composed of two composite groups $c_{i11}$ and $c_{i12}$, and an atomic group $a_{i13}$. Similarly, $c_{i11}$ and $c_{i12}$ are composite groups that contain atomic groups. Each atomic/composite group can have an optional compensation plan and/or contingency plan. Operation execution failure can occur on an operation at any level of nesting. The purpose of the DeltaGrid service composition model is to automatically resolve operation execution failure using compensation, contingency, and DE-rollback at different composition levels.
5. ENTITY EXECUTION SEMANTICS

As the building block of a process, the behavior of a DEGS operation impacts the state of the higher-level execution entities that are composed of DEGS operations. An operation may need backward recovery either because of its own execution failure, or due to another operation’s failure within the context of composite group execution. Under these two cases, a DEGS operation can present different recoverability options that affect the recovery activities of the entire process. This section defines the execution semantics of a DEGS operation and addresses recoverability of a DEGS operation.

As defined in the previous section, an operation is a DEGS service. The DeltaGrid system assumes that each DEGS that participates in the DeltaGrid service composition environment is an autonomous entity that guarantees its local correctness through a proper concurrency control mechanism, exposing serializability or variations of serializability to the service composition environment. Due to different functionality and implementation provided by service vendors, a DEGS operation presents one of the following transaction semantics:

1. A flat transaction or nested transaction. When an operation fails, the operation is automatically rolled back by the underlying database system. This type of DEGS operation is an ACID DEGS operation.

2. A multi-level transaction. A multi-level transaction is composed of multiple subtransactions. Each subtransaction is an ACID transaction that can unilaterally commit. If a multi-level transaction fails due to failure of a subtransaction, rollback usually cannot be applied since some subtransactions might have committed. Instead, a local compensating transaction is executed to restore the system to a consistent state. This type of DEGS operation is referred to as a multi-level DEGS operation. The local compensating transaction is atomic and preserves serializability as defined in the multi-level transaction model.

This section elaborates on recoverability issues for a DEGS operation. We first compare the execution semantics of an ACID DEGS operation and a multi-level DEGS operation. Since the

Figure 4. An Abstract View of a Sample Process

\[ p_1 = cg_1 \]

\[ cg_{11} \]

\[ ag_{111} \]

\[ op_{11} \]

\[ cop_{11} \]

\[ top_{11} \]

\[ ag_{112} \]

\[ op_{12} \]

\[ cop_{12} \]

\[ ag_{113} \]

\[ op_{13} \]

\[ top_{13} \]

\[ cg_{11}.cop \]

\[ cg_{11}.top \]

\[ cg_{12} \]

\[ ag_{121} \]

\[ op_{14} \text{ (non-critical)} \]

\[ ag_{122} \]

\[ op_{15} \]

\[ cop_{15} \]

\[ cg_{12}.top \]

\[ ag_{13} \]

\[ op_{16} \]

\[ cop_{16} \]

\[ top_{16} \]

\[ cg_1 \text{.cop} \]

\[ cg_1 \text{.top} \]
compensating transaction of a multi-level DEGS operation might fail, we then specify the recovery options for a multi-level DEGS operation for the case where the operation fails before the commit process. This case is referred to as **pre-commit recoverability**. After an operation commits, the failed execution of a subsequent operation within the same process can cause a successfully committed operation to be compensated. This case is referred to as **post-commit recoverability**. We also present the post-commit recovery options for a DEGS operation.

### 5.1 DEGS Operation Execution Semantics

Figure 5 compares the execution semantics of an ACID DEGS operation, shown in (a), with a multi-level DEGS operation, shown in (b). An ACID DEGS operation has four states: \{active, successful, failed, aborted\}. An operation enters the active state when it is invoked. If the execution successfully terminates, the operation enters a successful state, otherwise it enters a failed state. A failed state means that a runtime failure occurs during the operation execution that makes the execution invalid. An ACID DEGS operation can automatically roll back to enter an aborted state. This rollback activity is supported by the underlying database system and we assume it is always successful. Thus an ACID DEGS operation’s termination state is either successful or aborted.

![Figure 5. Transaction Semantics of a DEGS Operation](image)

As shown in Figure 5 (b), a multi-level DEGS operation has four states: \{active, successful, failed, compensated\}. The difference is that when a multi-level DEGS operation fails, a compensating transaction is executed, possibly containing multiple compensation steps for subtransactions that have been executed. This local compensating transaction is invoked automatically by the DEGS as a multi-level transaction processing system, independent of the DeltaGrid recovery capability. From the service composition point of view, this compensating transaction is conducted before the commit of the entire DEGS operation as a multi-level transaction, and is referred to as **pre-commit-compensation**.

Successful pre-commit-compensation leads to a compensated state. However in a realistic execution environment, a pre-commit-compensation might fail as well. As an atomic transaction, the failed pre-commit-compensation will be rolled back by the supporting transaction processing system and leaves no partial effect in the DEGS execution environment. So when a pre-commit-compensation fails, a multi-level DEGS operation remains in a failed state. Thus the termination states of a multi-level DEGS operation include: \{successful, compensated, failed\}. When pre-commit-compensation fails, another mechanism is needed to clean up the effect of the failed...
operation execution. The pre-commit-compensation is a failure recovery mechanism provided by a multi-level DEGS, and not by the DeltaGrid environment.

In the DeltaGrid system, an operation \( Op_i \) might need backward recovery if 1) \( Op_i \) fails before termination, or 2) \( Op_i \) successfully terminates, but another operation execution failure requires \( Op_i \) to be recovered in the context of composite group execution. In the former case, the recovery of \( Op_i \) happens before \( Op_i \) commits, which is referred to as pre-commit recovery. In the latter case, the recovery of \( Op_i \) happens after \( Op_i \) commits, which is referred to as post-commit recovery. In the following, we present the recovery capability of the operation in each case.

### 5.1.1 Pre-commit Recoverability

When the compensating transaction of a multi-level DEGS operation fails, the effect of the failed operation execution remains in the DEGS execution environment. Pre-commit recovery activities are applied to clean up the failed operation execution before the operation terminates.

**Definition 1 (Pre-commit Recoverability):** Pre-commit recoverability specifies how a DEGS operation should be recovered when an execution failure occurs before the DEGS operation as an execution unit commits.

Table 2 presents pre-commit recovery options for a DEGS operation. Ideally, a DEGS operation’s pre-commit recoverability is *automatic rollback* for an ACID DEGS operation, or *pre-commit-compensation* for a multi-level DEGS. Realistically, a pre-commit-compensation of a multi-level DEGS operation might fail. With the delta capture capability, a DEGS can reverse the effect of the original operation through *DE-rollback* if the recoverability conditions are satisfied. If DE-rollback cannot be applied due to the violation of the semantic conditions for DE-rollback, the service composition model requires a DEGS provider to offer a *service reset* function. The service reset function cleans up the effect of a failed operation and prepares the DEGS execution environment for the next service invocation. A service reset typically requires a special program or a human agent to resolve the failed operation execution.

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic rollback</td>
<td>The failed service execution can be automatically rolled back by a service provider</td>
</tr>
<tr>
<td>Pre-Commit-Compensation</td>
<td>A pre-commit-compensation is invoked by a service provider to backward recover a failed operation.</td>
</tr>
<tr>
<td>DE-rollback</td>
<td>A failed operation can be reversed by executing DE-rollback</td>
</tr>
<tr>
<td>Service Reset</td>
<td>The service provider offers a service reset function to clean up the service execution environment.</td>
</tr>
</tbody>
</table>

Figure 6 shows the state diagram of a DEGS operation with pre-commit recoverability. Compared with Figure 5, Figure 6 adds support for pre-commit-compensation failure handling and eliminates the failed state as a termination state. As shown in Figure 6, a DEGS operation enters the successful state when the operation successfully executes. If the operation fails, the operation transits to different states depending on whether the operation is an ACID DEGS operation or a multi-level DEGS operation. If an ACID operation fails, the operation enters the aborted state. Otherwise if a multi-level operation fails, the operation enters the compensated state when the pre-commit compensation succeeds. If the pre-commit compensation fails, the operation stays in the failed state.
When a pre-commit-compensation fails, the DeltaGrid system will initiate recovery activities. The first recovery option adopted by the DeltaGrid system is DE-rollback. The DeltaGrid system will determine if the semantic conditions for DE-rollback hold. If yes, DE-rollback is invoked, leading the operation to the DE-rollback state. Otherwise, the service reset function will be invoked, leading the operation to the service-reset state. By performing DE-rollback or service reset, a DEGS operation exits the failed state. DE-rollback and service reset as recovery activities are initiated by the DeltaGrid system. The pre-commit recovery activities, namely automatic rollback, pre-commit-compensation, DE-rollback, and service reset, transform a failed operation execution to a pre-commit recovered state, which represents one of four concrete states: aborted, compensated, DE-rollback, service-reset. Thus the termination state of a DEGS operation is either successful or pre-commit recovered.

![Diagram of DEGS Operation with Pre-Commit-Compensation Failure Considered]

**5.1.2 Post-Commit Recoverability**

Contrary to pre-commit recoverability, which defines how to clean up a failed operation execution, post-commit recoverability specifies how to semantically undo the effect of a successfully terminated operation due to another operation’s execution failure. This section defines post-commit recoverability of a DEGS operation and discusses different post-commit recoverability options.

**Definition 2 (Post-commit Recoverability):** Post-commit recoverability specifies how an operation’s effect can be semantically undone after the operation successfully terminates.

Post-commit recoverability is considered when a completed operation inside of a composite group needs to be undone due to runtime failure of another operation. Table 3 defines three post-commit recoverability options: reversible (through DE-rollback), compensatable, or dismissible. Since post-commit recovery is only applicable in the context of composite group execution, the
invocation condition of post-commit recovery is addressed in Section 5.3 on Composite Group Execution Semantics.

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

5.2 Atomic Group Execution Semantics

An atomic group (ag) maximizes the success of an operation execution by providing a contingency plan. If necessary, an ag can be semantically undone by executing a post-commit recovery activity, such as compensation or DE-rollback, depending on the primary operation’s post-commit recoverability.

With relaxed atomicity, the success of a process execution can be application-dependent and might not require every operation to be successfully executed. The DeltaGrid service composition model offers the flexibility of marking execution entities where failure does not affect the execution of the enclosing composite group using a criticality decorator. By default, an operation’s post-commit recoverability is compensatable.

**Definition 3 (Criticality):** An atomic group is critical if its successful execution is mandatory for the enclosing composite group. A non-critical group indicates that the failure of this group will not impact the state of the enclosing composite group, and the composite group can continue execution. When runtime execution failure occurs, contingency must be executed for critical groups, while contingency is not necessary for a non-critical group. By default, a group is critical.

As an example, in Figure 4, if ag_{121} fails, cg_{12} will continue executing since ag_{121} is non-critical. Thus in the specification, there is no need to define a compensation and contingency plan for ag_{121}. Figure 7 describes the execution semantics of an atomic group ag. An ag enters the active state if the primary operation is invoked. If the primary operation successfully terminates, the primary operation enters the successful state, as described in Figure 6, leading ag to enter the ag successful state. If the primary operation fails, ag enters the pre-commit recovered state, as defined in Figure 6.

From the pre-commit recovered state, contingency will be executed if the atomic group is critical, as a transition initiated by the DeltaGrid system. Similar to a pre-commit-compensation, the contingency is enforced as an atomic transaction in a DEGS. If the contingency succeeds, the ag enters the ag successful state. Otherwise, the contingency is aborted, which leads the ag to the ag aborted state. If the atomic group is non-critical, ag enters the ag aborted state. The termination state of an atomic group is either ag successful or ag aborted. Compensation and DE-rollback as post-commit recovery techniques for an atomic group are addressed in the context of composite group execution semantics as described in the following subsection.
5.3 Composite Group Execution Semantics

The recoverability of a composite group can be defined using the concepts of shallow compensation and deep compensation. The terms shallow and deep compensation were originally defined in (Laymann, 1995). Our research extends these concepts for use with nested service composition.

**Definition 4 (Shallow Compensation):** Assume a composite group \( cg_k \) is defined as \( cg_k = <(ag_{km} \mid cg_{kn}), cop_k [\top op_k]> \). Shallow compensation of \( cg_k \) is the invocation of the compensation operation defined for the composite group \( cg_k \) which is \( cop_k \).

**Definition 5 (Deep Compensation):** Assume a composite group \( cg_k \) is defined as \( cg_k = <(ag_{km} \mid cg_{kn}), cop_k [\top op_k]> \). Within the context of a composite group \( cg_k \), a subgroup is either an atomic group defined as \( ag_{km} = <op_{km}, cop_{km} [\top op_k]> \), or a composite group defined as \( cg_{kn} = <(ag_{knm} \mid cg_{kn}), cop_{kn} [\top op_{kn}]> \). Deep compensation of \( cg_k \) is the invocation of post-commit recovery activity (compensation or DE-rollback) for each executed subgroup within the composite group, such as \( cop_{km} \) for an atomic group, and \( cop_{kn} \) for a nested composite group.

Shallow compensation is invoked when a composite group successfully terminates but needs a semantic undo due to the failure of another operation execution. A deep compensation is invoked if: 1) a composite group fails due to a subgroup execution failure, and needs to trigger the post-commit recovery of executed subgroups, or 2) a composite group successfully terminates, but no shallow compensation is defined for the composite group.

As a backward recovery mechanism for a successfully executed composite group, shallow compensation has higher priority than deep compensation. For example, in Figure 4, the failure of a critical subgroup \( ag_{13} \) (both \( op_{15} \) and \( top_{15} \) fail) within the enclosing composite group \( cg_1 \) causes the two executed composite groups \( cg_{11} \) and \( cg_{12} \) to be compensated. Since \( cg_{11} \) has a pre-defined shallow compensation, the shallow compensation \( cg_{11}.cop \) will be executed. \( cg_{12} \)'s deep compensation will be invoked since \( cg_{12} \) does not have shallow compensation.
Figure 8 (a) presents the execution semantics of a composite group $cg$ composed of only atomic subgroups, denoted as $cg = \langle ag_1^k \cdot \left[ \right., \cop, \left[ \right. \top \rangle \rangle$. $cg$ remains active during a subgroup’s execution. If all the subgroups terminate successfully, $cg$ enters the $cg$ successful state. If a particular subgroup $ag_k$ fails, $ag_k$ enters the $ag_k$ aborted state. $cg$ then enters different states depending on whether $ag_k$ is the first subgroup of $cg$. If $ag_k$ is the first subgroup of $cg$, the pre-commit recovery of $ag_k$ leads $cg$ to enter the $cg$ aborted state. Otherwise all of the previously executed subgroups $(ag_{1..k-1})$ will be post-commit recovered, leading $cg$ to the $cg$ deep compensated state. To simplify the state diagram, the $cg$ extended abort state is introduced to represent either the $cg$ aborted state or the $cg$ deep compensated state. The $cg$ extended abort state indicates that the partial result of a composite group $cg$’s execution has been cleaned up, and the contingency for the composite group can be executed.

From the $cg$ extended abort state, $cg$’s contingency can be executed. If the contingency succeeds, $cg$ enters the $cg$ successful state. If the contingency fails, the contingency rolls back as an atomic transaction, and $cg$ remains in the $cg$ extended abort state.

In Figure 8 (a), the $cg$ deep compensated state can be reached only if every step during a deep compensation process succeeds. In another words, compensation of every executed subgroup ($ag_{1..k-1}$) must succeed. Realistically a subgroup’s compensation might fail when a subgroup’s compensation fails during a deep compensation, DE-rollback or the service reset function will be applied, in the same manner as a pre-commit-compensation failure is handled. DE-rollback is the first recovery option when a subgroup’s compensation fails. If the semantic condition for DE-rollback holds, DE-rollback can be invoked to reach the $cg$ deep compensated state. If not, the service reset function will be invoked as the second recovery option.

Figure 8 (b) presents the execution semantics of a composite group $cg$ composed of subgroups $sg_k$ that can be either atomic groups or composite groups, denoted as $cg = \langle sg_1^k \cdot \left[ \right., \cop, \left[ \right. \top \rangle \rangle$. Similar to Figure 8 (a), $cg$ remains in the active state when a subgroup is executing. If all the subgroups succeed, $cg$ enters the $cg$ successful state. Otherwise, if any subgroup $sg_k$ fails, $sg_k$ enters the $sg_k$ extended abort state, as defined in Figure 7 (if $sg_k$ is an atomic group) and in Figure 8 (a) (if $sg_k$ is a composite subgroup). Depending on whether $sg_k$ is the first subgroup of $cg$, $cg$
enters either the cg aborted state or the cg deep compensated state, following the same transition defined in Figure 8 (a). The state transition caused by the contingency execution is the same as that presented in Figure 8 (a).

5.4 Backward Recovery of an Atomic Group and a Composite Group

Figure 9 presents the backward recovery semantics of an atomic group. A completed atomic group might need a backward recovery caused by the execution failure of another entity during process execution. Backward recovery of an atomic group ag cancels the effect of a successfully executed atomic group, thus backward recovery of ag starts from the state ag successful in Figure 7. An atomic group has three backward recovery options: compensation, DE-rollback, or service reset, which are invoked based on post-commit recoverability of the primary operation. If the primary operation is compensatable, compensation is invoked. If compensation succeeds, ag enters the ag compensated state. If compensation fails, ag remains in the ag successful state since the effect of compensation will be removed by the DEGS execution environment. When compensation fails, DE-rollback can be invoked if DE-rollback is applicable, which leads ag to the ag DE-rollback state. If DE-rollback is not applicable, service reset can be performed, leading ag to the ag service-reset state. Thus the execution of backward recovery activity will leave ag in the ag post-commit recovered state, which represents one of three concrete states: ag compensated, ag DE-rollback, or ag service-reset.

Figure 10 (a) presents the backward recovery semantics of a composite group that is composed of only atomic groups. The backward recovery of a composite group cg starts from the cg successful state of Figure 8 (a). If cg has a shallow compensation and the shallow compensation succeeds, cg enters the cg shallow compensated state. If cg does not have shallow compensation, or if shallow compensation fails, deep compensation can be executed on cg by executing backward recovery activity for each enclosed atomic group agk. According to the backward recovery...
semantics of an atomic group presented in Figure 9, backward recovery of agk is guaranteed to terminate in the agk post-commit recovered state, thus lead cg to the cg deep compensated state.

Figure 10. Composite Group Backward Recovery Semantics

Figure 10 (b) presents the backward recovery semantics of a composite group cg that is composed of atomic groups and/or nested composite groups and/or nested subgroup sgik. As in Figure 10 (a), shallow compensation is invoked first if available. When deep compensation is needed, there are several possible intermediate states based on sgik. If sgik is an atomic group, backward recovery of sgik leads sgik to the sgik post-commit recovered state. If sgik is a composite group, sgik terminates in the sgik shallow compensated state or the sgik deep compensated state, as shown in Figure 10 (b). Thus every subgroup is successfully backward recovered, leading cg to the cg deep compensated state.

6. RECOVERY ALGORITHM

This section presents algorithms for the recovery of a failed operation execution in the context of a process composed of nested composite groups. The recovery process requires deep compensation of a composite group, which will eventually invoke the post-commit recovery of atomic groups. The algorithm is divided into three main steps: 1) the atomic group post-commit recovery algorithm for backward recovery of an atomic group after the successful execution of its primary operation, 2) the composite group deep compensation algorithm, which invokes the post-commit recovery of executed atomic subgroups, and 3) the top-level algorithm for recovery of a failed operation execution within the context of process execution.

6.1 Atomic Group Post-Commit Recovery Algorithm

An atomic group ag is complete if ag contains necessary compensation and contingency plans for its primary operation op, depending on op’s post-commit recoverability and ag’s criticality in the context of the immediately enclosing composite group. A complete non-critical atomic group only contains the primary operation, while a complete critical atomic group requires a contingency plan. The post-commit recoverability determines if a compensation plan is needed for a complete critical atomic group. If the primary operation is compensatable, the atomic group

---

This is a continuation of the document text, providing a natural representation of the content. The diagram is not included in the text. The text is focused on the recovery algorithm, specifically the atomic and composite group recovery semantics and the division of the recovery process into three main steps: atomic group post-commit recovery, deep compensation of composite groups, and top-level recovery algorithm.
should contain a compensation plan. If the primary operation is reversible or dismissible, no compensation is needed. However, in a process definition, an atomic specification might be incomplete if compensation is required but not provided by the service provider. Under this circumstance, DE-rollback or service reset will be invoked, depending on whether the semantic conditions for DE-rollback hold.

Figure 11 outlines the algorithm to determine whether to invoke DE-rollback or service reset on an operation. The input to the procedure is a failed operation to be backward recovered. After the execution of the procedure, the effect of the given operation is eliminated either through DE-rollback or service reset. The algorithm begins with checking if the semantic conditions for DE-rollback of the given operation holds. If yes, DE-rollback will be invoked. Otherwise service reset will be called on a DEGS. This procedure is used in the atomic group post-commit recovery algorithm when compensation is not available, or when compensation fails.

Figure 11. Procedure to Invoke DE-Rollback or Service Reset on an Operation

Figure 12 presents the algorithm to recover an atomic group after its successful termination based on the atomic group’s criticality and its primary operation’s post-commit recoverability specification. The input to the algorithm is an atomic group to be post-commit recovered. After execution of the algorithm, the given atomic group is post-commit recovered through compensation, DE-rollback, or service reset.

The atomic group post-commit recovery algorithm corresponds to the recovery semantics of Figure 9, starting the backward recovery of $ag_i$ from the state $ag_i$ successful (the end state of Figure 7 and the start state of Figure 9). The successful execution of compensation (case 1.1.1) leads $ag_i$ to the state $ag_i$ compensated. If compensation fails (case 1.1.2), compensation is not defined for $ag_i$ (case 1.2), or compensation is not necessary (case 2), DE-rollback or service reset will be invoked, which leads $ag_i$ to the state $ag_i$ DE-rollback or $ag_i$ service-reset. In case 3, $ag_i$ does not need a backward recovery. As a result, the atomic group post-commit recovery algorithm guarantees $ag_i$ to be in the state $ag_i$ post-commit recovery of the state diagram in Figure 9, if any backward recovery activity is necessary.

As an example, an atomic group is defined as $ag_i = <op_i, [cop_i], [top_i]>$. The algorithm first checks the post-commit recoverability of the given atomic group’s primary operation $op_i$. There are three possible options based on $op_i$’s post-commit recoverability:

1) $op_i$ is compensatable. If $ag_i$ has compensation $cop_i$, $cop_i$ will be invoked. If $cop_i$ succeeds, $ag_i$ is compensated and the algorithm returns. Otherwise if $cop_i$ fails, the algorithm will invoke DE-rollbackOrServiceReset($op_i$) presented in Figure 11 to recover $op_i$ through DE-rollback or service reset.

2) $op_i$ is reversible. There will be no compensation in the atomic group. DE-rollbackOrServiceReset($op_i$) is invoked to recover the atomic group.

3) $op_i$ is dismissible. No recovery action is needed in this case.

```java
public void DE-rollbackOrServiceReset(String op)
{
    //check if DE-rollbackOrServiceReset is defined for op
    if (compensationDefined(op))
    {
        compensation(op);
        return;
    }
    else
    {
        //if compensation is not defined, invoke DE-rollback or service reset
        DE-rollbackOrServiceReset(op);
    }
}
```
6.2 Composite Group Deep Compensation Algorithm

As discussed previously, deep compensation of a composite group \( \text{cgi} \) is executed either due to a subgroup execution failure before \( \text{cgi} \) completes, or due to an operation failure outside of \( \text{cgi} \) after \( \text{cgi} \) completes but \( \text{cgi} \) has no shallow compensation.

In the context of composite group execution, the failure of an atomic group \( \text{agik} \) means that the atomic group’s primary operation and the contingency plan (if it exists) fail, and the atomic group ends in the \( \text{ag aborted} \) state. If the atomic group is non-critical, failure of the group will not affect the execution of the enclosing composite group \( \text{cgi} \). However, if a critical atomic group \( \text{agik} \) fails and the immediately enclosing composite group \( \text{cgi} \) is critical, \( \text{cgi} \) needs to be deep compensated.

Figure 13 presents an algorithm to deep compensate a composite group \( \text{cgi} \). This algorithm recursively invokes the deep compensation of an immediately enclosing composite group of a subgroup if the contingency of the subgroup fails. The input to the algorithm is the composite group to be deep compensated. After the execution of the algorithm, the effect of \( \text{cgi} \) is semantically undone by invoking post-commit recovery of executed subgroups in reverse execution order, if a subgroup is critical to \( \text{cgi} \)’s execution.

The algorithm begins with getting a list of executed critical subgroups \( \text{sgi} \) in reverse execution order. The algorithm then iterates through each subgroup \( \text{sgi} \). If \( \text{sgi} \) is an atomic group, \( \text{post-commitRecoverAtomicGroup(sgi)} \) will be invoked. Otherwise if \( \text{sgi} \) is a composite group and \( \text{sgi} \) has...
shallow compensation \( csg_k \), \( csg_k \) will be executed. If \( sg_k \) does not have shallow compensation or shallow compensation fails, the algorithm will recursively invoke deep compensation on \( sg_k \).

```java
public void deepCompensate(CompositeGroup cg) {
    //get a list of executed critical subgroups of cg in reverse execution order
    C = \{ sg_k | sg_k \in cg \} (k = n \rightarrow 1)

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

**Figure 13. Composite Group Deep Compensation Algorithm**

As we have discussed, the deep compensation of a composite group \( cg \) is invoked if 1) a critical subgroup of \( cg \) fails before \( cg \) completes, or 2) \( cg \) successfully completes, \( cg \) does not have a pre-defined shallow compensation, and \( cg \) needs to be backward recovered due to the failure of another operation execution. In either case, the deep compensation process contains the post-commit recovery of executed critical subgroups. For example, in Figure 4, if \( ag_{112} \) fails, \( ag_{111} \) will be compensated, which is the deep compensation of the enclosing composite group \( cg_{11} \) before \( cg_{11} \) completes. If \( ag_{13} \) fails, \( cg_{12} \) and \( cg_{11} \) will be compensated, which is the deep compensation of \( cg_{1} \) before \( cg_{1} \) completes. \( cg_{12} \) needs to be deep compensated by executing \( ag_{122} \) since \( cg_{12} \) does not have a shallow compensation. \( cg_{11} \) can be shallow compensated by executing \( cg_{11}.cop \).

The composite group deep compensation algorithm leads a completed composite group \( cg \) to the state \( cg \) deep compensated, conforming to the backward recovery semantics shown in Figure 10 (b). The algorithm starts with \( cg \) in the state \( cg \) successful. If a subgroup \( sg_k \) is an atomic group (case 1), \( sg_k \) enters the state \( sg_k \) post-commit recovered, by executing the post-commit recovery algorithm for \( sg_k \). If \( sg_k \) is a composite group and shallow compensation is available (case 2.1.1), the successful invocation of shallow compensation leads \( sg_k \) to the state \( sg_k \) shallow compensated. If shallow compensation fails (case 2.1.2), or shallow compensation is unavailable (case 2.2), the
deep compensation algorithm will be invoked on $sg_k$, which leads $sg_k$ to the state $sg_k$ deep compensated. Thus the execution of the algorithm guarantees $cg_i$ to enter one of the concrete states of the state $cg_i$ deep compensated, as shown in Figure 10 (b).

If $cg_i$ needs a deep compensation due to a subgroup failure before its completion, the composite group deep compensation algorithm starts from the state $sg_i$ extended abort in Figure 8 (b). Every executed subgroup performs a backward recovery, which leads $cg_i$ to the state $cg_i$ extended abort in Figure 8 (b). As a summary, the execution of the composite group deep compensation algorithm leads a composite group $cg_i$ to the state $cg_i$ deep compensated if $cg_i$ successfully completed prior to compensation, or to the state $cg_i$ extended abort if $cg_i$ is in the process of execution when failure occurs. In either case, the algorithm execution conforms to the state transitions shown in Figures 8 and 10.

6.3 Operation Execution Failure Recovery Algorithm

Figure 14 presents the operation execution failure recovery algorithm which recovers a failed operation execution in the context of a process execution. The input to the algorithm is a failed operation $op_i$. The output of the algorithm is a Boolean value indicating whether the process with an operation execution failure can be forward recovered or not. If the method returns true, the process has been recovered and can continue with the next execution entity. Returning false means that the entire process has been backward recovered.

The algorithm first gets the enclosing atomic group $ag_i$ of the failed operation $op_i$. Then the algorithm checks if $ag_i$ is critical. If $ag_i$ is not critical, the algorithm returns true. If $ag_i$ is critical and $ag_i$ has a contingency $top_i$, $top_i$ is invoked. If $top_i$ succeeds, the algorithm returns true. However if $top_i$ fails or $ag_i$ has no contingency $top_i$, the fault will be propagated to the immediately enclosing composite group of $ag_i$ by executing a procedure $propagateFailure(\text{ag}_i)$. The procedure $propagateFailure(\text{ag}_i)$ will recover the failed atomic group within the scope of the enclosing composite group.

Figure 15 presents the algorithm for handling the propagation of a failed atomic group in the context of nested composite group execution. The input to the algorithm is a failed atomic group $ag_i$. The output is a Boolean value indicating whether failure of the atomic group can be forward recovered. If the method returns true, the enclosing process of the atomic group has been recovered. Otherwise the process is backward recovered.

As shown in Figure 15, the first step of the atomic group failure propagation algorithm is obtaining the immediate enclosing composite group $cg_i$ of the failed atomic group $ag_i$. Then the algorithm checks if $cg_i$ is critical. If $cg_i$ is not critical, the algorithm returns true. Otherwise, $deepCompensate(cg_i)$ is invoked followed by checking the availability of $cg_i$’s contingency. If $cg_i$ has contingency $top_i$ and $top_i$ succeeds, the algorithm returns true. However if $cg_i$ has no contingency $top_i$ or if $top_i$ fails, the fault is propagated to the immediate enclosing composite group $cg_i$, then the process of recovering the $cg_i$ starts again. The fault propagation repeats until either the contingency of a composite group succeeds, or the top-level composite group (the process) is reached. In the former case, the process is successfully forward recovered, and can continue with the next execution entity. In the latter case, the entire process is backward recovered.

The operation failure recovery algorithm starts the recovery from the state pre-commit recovered for operation $op_i$ in Figure 7. If the enclosing atomic group $ag_i$ is non-critical (case 1), no contingency is necessary, and $ag_i$ enters the state $ag$ aborted. If $ag_i$ is critical and contingency succeeds (case 2.1.1), $ag_i$ enters the state $ag$ successful. In either case, the algorithm returns true and the enclosing process continues with the next operation execution. However if the contingency fails (case 2.1.2), or contingency is unavailable (case 2.2), the failure of $ag_i$ is propagated to the enclosing composite group by invoking the atomic group failure propagation algorithm $propagateFailure$ in Figure 15.
public boolean recover(Operation opij) {
    //get the enclosing atomic group of opij
    //check if agij is critical
    CASE:
        1. agij is non-critical:
            RETURN TRUE;
        2. agij is critical:
            //check if agij has contingency
            CASE:
                2.1 agij has a contingency:
                    EXE:
                    //check if sgik has contingency
                    CASE:
                        2.2 agij has no contingency:
                            EXE:
                            END CASE;
                        2.3 agij has a contingency:
                            END CASE;
                    END CASE;
                2.2 agij has no contingency:
                    EXE:
                    END CASE;
            END CASE;
    END CASE;
}

Figure 14. Operation Failure Recovery Algorithm

The algorithm propagateFailure starts the recovery of agij from the state ag aborted in Figure 8 (a). If the immediately enclosing composite group cgij of agij is non-critical (case 1), propagateFailure returns true and the process continues with the next step. Otherwise, deep compensation is invoked on cgij, leading cgij to the state cgij extended abort. If cgij has a contingency and contingency succeeds (case 2.1.1), cgij enters the state cgij successful. The algorithm returns true and the enclosing process continues with the next operation execution. However if cgij has no contingency (case 2.2) or contingency fails (case 2.1.2), cgij remains in the state cgij extended abort, which is sgik extended abort in Figure 8 (b), where propagateFailure starts recursive invocation. The failure of sgik causes the immediate enclosing composite group cgij to be deep compensated, and contingency executed if available. If contingency succeeds (case 2.1.1), cgij enters the state cgij successful, the algorithm returns true, and the process continues with the next operation. If contingency is unavailable (case 2) or contingency fails (case 2.1.2), the algorithm is again recursively invoked until cgij reaches the top level composite group (the process). If the top level composite group has a contingency and the contingency succeeds, the process enters the state cgij successful. Otherwise the process enters the state cgij deep compensated state and ends. In summary, the invocation of the operation failure recovery algorithm either forward recovers a process so the process can continue with the next operation, or backward recovers a process, thus a process is deep compensated, conforming to the state transitions shown in Figures 7, 8, and 10.

Using the process defined in Figure 4 as an example, when op15 fails, since agij does not have a contingency plan, the enclosing composite group cgij will be deep compensated. Since agij is a non-critical subgroup of cgij and requires no post-commit recovery, no action is invoked as the deep compensation procedure of cgij. The deep compensation will be followed by cgij’s contingency. If cgij’s contingency succeeds, the process will continue with agij. However if cgij’s contingency fails, the enclosing composite group of cgij, cg1, will be deep compensated. The deep compensation of cgij first invokes the shallow compensation of cgij. If cgij’s shallow compensation
fails, $c_{g_{i1}}$ will be deep compensated. After $c_{g_{i1}}$'s deep compensation, $c_{g_{i1}}$'s contingency will be executed. If the contingency is successful, the process terminates successfully. Otherwise, the process is backward recovered.

Figure 15. Atomic Group Failure Propagation Algorithm

```
public boolean propagateFailure(AtomicGroup ag_{ij})
{
    get the enclosing composite group of $ag_{ij}$ => cg;
    //check if cg exists
    WHILE (cg is not NULL)
    {
        //check if cg is critical
        CASE:
            1. cg is non-critical:
            RETURN TRUE;
            2. cg is critical:
            EXECUTE deepCompensation;
            //check if cg has contingency
            CASE:
                1. cg has top:
                EXECUTE compensates;
```

7. CASE STUDY

This section introduces a placeClientOrder process in the context of an online shopping application to illustrate the use of the service composition and recovery model. The online shopping application contains typical business processes that describe the activities conducted by shoppers, the store and vendors. For example, the process placeClientOrder is responsible for invoking services that place client orders and decrease the inventory quantity.

Figure 16 presents a graphical view of the placeClientOrder process, using the same notation as the abstract process example presented in Figure 4. As shown in Figure 16, the process placeClientOrder is hierarchically composed of composite groups and atomic groups. An atomic group has an operation, an optional compensation (cop) and contingency (top).

The placeClientOrder process starts when a client submits a client order by invoking a DEGS operation receiveClientOrder. The next operation creditCheck verifies if the client has a good credit
standing to pay for the order. If the client passes the creditCheck, the inventory will be checked to see if there are sufficient inventory items to fill the order by executing checkInventory. If the client does not pass the credit check, the order will be rejected. If there are sufficient inventory items, the operation chargeCreditCard is to be executed to charge the client’s credit card, and the operation decInventory is executed to decrease inventory. These two operations are grouped into a composite group indicating that both operations should be successfully executed as a unit. Then the order will be packed through operation packOrder and shipped through operation upsShipOrder. If the inventory is not sufficient to fill the order, the order will be marked as a backorder through operation addBackorder, and the client will be charged the full amount.

Figure 16. placeClientOrder Process Definition

When there is a service execution failure during process execution, the process will be recovered based on the recovery specification embedded in the process definition, such as compensation and contingency, as well as the recovery semantics of the service composition and recovery model. For example, if operation upsShipOrder fails, the contingency fedexShipOrder will be invoked, sending the order package through Fedex instead of UPS. If a client requests to cancel the order after the operation packOrder but before upsShipOrder, each executed operation will be backward recovered in the reverse execution order using the following list of recovery commands: [cop:unpackOrder, cop:incInventory, cop:creditBack, DE-rollback:checkInventory, DE-rollback:checkCredit, cop:chgOrderStatus]. DE-rollback is to be performed on operations checkInventory and checkCredit since these two operations do not have pre-defined compensation
and no other concurrently executing processes are write dependent on these two operations. Furthermore, since these two operations do not modify any data, no recovery actions will be performed for these two operations. Thus the final recovery commands for cancellation of an order is: \{cop:unpackOrder, cop:incInventory, cop:creditBack, cop:chgOrderStatus\}.

8. SIMULATION AND EVALUATION OF THE COMPOSITION AND RECOVERY MODEL

The abstract model provides a theoretical foundation for building a semantically robust execution environment for processes that execute over distributed DEGSs. To support the concepts and algorithms defined in the composition and recovery model, this research has designed and implemented a DeltaGrid simulation framework, using DEVSJAVA (Zeigler and Sarjoughian 2004), a Java-based modeling and simulation tool for discrete event system specification.

Figure 17 shows the major components of the simulation framework, including fully implemented components for the PHCS and the Process Recovery System (PRS), as well as simulated components for DEGS and the execution engine. To support the correctness of multiple process execution, our research has also revised the recovery algorithms presented in this paper to support the recovery of concurrently executing processes that are affected by the recovery of a failed process (Xiao, 2006). The recoverQ in Figure 17 is a component that is used to schedule recovery operations for concurrently executing processes. The work in (Xiao, 2006; Xiao and Urban 2007a) describes our results with identification and recovery of concurrent processes that are either read or write dependent on a failed process.

The execution engine invokes recovery algorithms implemented in the PRS to generate recovery commands for an operation execution failure, and invoke these recovery commands on DEGSs through the recoverQ. The execution engine also generates process execution context such as start time, end time, and execution state. DEGSs execute service operations and
associated recovery activities, such as contingency, compensation, DE-rollback, and service reset. After each operation execution, a DEGS generates a delta file, and updates the operation execution context. The PHCS receives delta files from DEGSs and builds the process execution history, integrating the deltas and process/operation execution context. The PRS generates the recovery commands to recover a process when a service execution failure occurs, implementing the recovery algorithms introduced in Section 6.

We developed different scenarios to demonstrate that the PRS can generate recovery commands for a failed process according to the specification of the service composition and recovery model. The execution scenarios also show the coordination among the execution engine, DEGSs, the PHCS, and the PRS in the context of single process failure recovery.

Figure 18 presents one test scenario of an abstract process. A process $p_1$ is the top level composite group $cg_1$. $p_1$ is composed of two composite groups $cg_{11}$ and $cg_{12}$, and an atomic group $ag_{13}$. Similarly, $cg_{11}$ and $cg_{12}$ are composite groups that contain atomic/composite groups. Each atomic/composite group can have an optional compensation plan and/or contingency plan. Operation execution failure can occur on an operation at any level of nesting.

This process is chosen to demonstrate the operation failure recovery since this process contains the following important cases for a process recovery: non-critical group ($ag_{121}$), atomic group without contingency ($ag_{112, ag_{122}}$), atomic group without compensation and contingency ($ag_{113}$), nested composite group ($cg_{11}$), and composite group without shallow compensation ($cg_{111, cg_{112}}$). The recovery procedure for the operation $op_{16}$ covers all the above cases. Thus in the simulation run, we inject operation execution failure on $op_{16}$ to demonstrate the capability of the recovery algorithm in handling the different cases listed above.

To run the scenario, we start a process instance $p_1$ as defined in Figure 18. We conducted several simulation runs and each time injected operation execution failure on different operations. We expect the following result from this set of simulation runs:

1. The PRS generates the recovery commands for operation execution failure according to the semantics defined in the service composition model.
2. When operation execution failure occurs, the execution engine suspends the failed process. The execution engine then invokes the PRS to generate recovery commands, and add these recovery commands to the recovery queue. After the recovery activities in the recovery queue are executed, forward execution activity in the suspended queue is added to the execution queue.

Table 4 presents the failure recovery commands generated by the PRS when the process fails at different operations in simulation runs. After the operation failure is recovered within the scope of a composite group, the process continues its execution from the operation at the same level of recovered composite group, indicated by the forward execute activity column in Table 4. The discussion column of Table 4 justifies why the recovery commands are correct in each case.

We also evaluated the PRS with respect to performance. Recovery command generation time is affected by two factors: 1) the number of concurrent processes ($n$) since $n$ affects the time to evaluate the applicability of DE-rollback and the time to retrieve an operation execution context; 2) the nesting level of a process, which represents the complexity of a process’s structure. The evaluation was conducted by varying concurrency and process complexity. To vary the number of concurrent processes, we tested on two different ranges: 10–100 processes (medium) and 100–1000 processes (large). To vary a process’s complexity, we tested on processes with nesting levels from 1-5. Figure 19 indicates that recovery command generation time has a linear increase when the number of concurrent processes grows. Our experiments showed that a flat process (with nesting level 1) required less than a millisecond of recovery command generation time, no matter how many concurrent processes are running. When the process nesting level is greater than 1, there is a 13% increase in processing time, under medium and large levels of concurrency. This increase is because of the need to increase write dependency retrieval time for nested processes. This set of results shows the recovery command generation for the failed process only.
As an extension of single process recovery presented in this paper, multiple process recovery command generation time is evaluated and discussed in (Xiao 2006, Xiao and Urban 2007b).

Figure 18. A Process Definition to Demonstrate the Functionality of the Recovery Algorithm

We also evaluated read/write dependency retrieval time. This evaluation is important for single process recovery since write dependency is needed to determine whether DE-rollback is applicable when compensation is not provided. Multiple process recovery performance is largely dependent on read/write dependency retrieval time due to the need to construct process dependency graphs. Our result shows that when the number of concurrent processes falls into two ranges: medium (10~100) and large (100~1000), read and write dependency retrieval time increases exponentially when the number of concurrent process grows. However when we optimize write dependency retrieval by segmenting the global delta object schedule into several smaller pieces and merge the result, linear increase has been achieved, as shown in Figure 20.
Table 4. Operation Execution Failure Recovery

<table>
<thead>
<tr>
<th>Failed operation</th>
<th>Backward recovery activity</th>
<th>Forward execution activity</th>
<th>Next operation</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td>op11</td>
<td>none</td>
<td>t&lt;sub&gt;p&lt;/sub&gt;11</td>
<td>op17</td>
<td>Contingency is available, execute contingency, continue p₁ with the next operation op₁7</td>
</tr>
<tr>
<td>op17</td>
<td>none</td>
<td>t&lt;sub&gt;p&lt;/sub&gt;17</td>
<td>op12</td>
<td>Contingency is available, execute contingency, continue p₁ with the next operation op₁2</td>
</tr>
<tr>
<td>op12</td>
<td>[cop₁₂, cop₁₇, cop₁₁]</td>
<td>cg₁₁.top</td>
<td>op₁₄</td>
<td>Contingency is not available, deep compensate cg₁₁, continue p₁ with the next operation op₁₄</td>
</tr>
<tr>
<td>op₁₃</td>
<td>[dop₁₃, cop₁₂, cop₁₇, cop₁₁]</td>
<td>cg₁₁.top</td>
<td>op₁₄</td>
<td>Compensation and contingency are not available, DE-rollback op₁₃, compensate op₁₂, deep compensate cg₁₁, continue p₁ with the next operation op₁₄</td>
</tr>
<tr>
<td>op₁₄</td>
<td>None</td>
<td>op₁₅</td>
<td>op₁₆</td>
<td>No need to recover a non-critical group, continue p₁ with the next operation op₁₅</td>
</tr>
<tr>
<td>op₁₅</td>
<td>[cop₁₅]</td>
<td>cg₁₂.top</td>
<td>op₁₆</td>
<td>Deep compensate cg₁₂, execute contingency of cg₁₂, continue p₁ with the next operation op₁₆</td>
</tr>
<tr>
<td>op₁₆</td>
<td>[cop₁₅, dop₁₃, cop₁₂, cop₁₇, cop₁₁]</td>
<td>cg₁₁.top</td>
<td>None</td>
<td>op₁₆ does not have contingency, deep compensate cg₁, and execute the contingency of cg₁, p₁ is forward recovered.</td>
</tr>
</tbody>
</table>

Figure 19. Recovery Command Generation Time
9. SUMMARY AND FUTURE DIRECTIONS

This research has defined a service composition and recovery model to support flexible, automatic process recovery in a loosely-coupled, distributed service composition environment. Different from existing solutions, the model provides a flexible hierarchical composition structure. Forward recovery is maximized through compensation or DE-rollback at different granularity levels, followed by contingency plans. The model includes detailed recovery algorithms to generate recovery commands whenever a service fails in the context of a process execution. This research has included the implementation of the DeltaGrid simulation framework to verify and support the service composition and recovery model, including an evaluation of the performance of the recovery command generation process.

This research has been extended to support multiple process recovery in a concurrent process execution environment, where the recovery of a failed process can cause read dependent or write dependent processes to also invoke recovery activities based on user-defined semantic conditions. When one process determines that it needs to execute compensating procedures, data changes introduced by compensation or DE-rollback of a process might affect other concurrently executing processes that have either read or written data that have been produced by the failed process. We refer to this situation as process interference. A robust service execution environment should recover a failed process and effectively handle process interference based on data dependencies and application semantics. To support correctness of multiple process execution, our research has revised the recovery algorithms presented in this paper to support the recovery of concurrently executing processes that are affected by the recovery of a failed process (Xiao, 2006; Xiao and Urban, 2007).

The performance of multi-process recovery is heavily dependent on read/write dependency retrieval in PHCS. When there is large number of processes in the environment, the PHCS can become a bottleneck, even with optimization. One of our future directions is to design a distributed PHCS system where the global delta object schedule is distributed over several sites and only related dependency information is transferred between sites. This can significantly reduce read/write dependency construction and retrieval time and improves system performance.

Another future direction is to study a more dynamic approach to service composition environment and revise the service composition model to support the use of events and rules. Events are needed for responding to exceptional conditions and to application exceptions, while rules are needed to define more flexible ways of responding to event. In many business processes, application exceptions frequently occur and require manual processing based on process execution status and/or read and write dependencies on other processes. Adding rule-based application exception handling will provide a more complete notion of process failure recovery for business processes that execute over distributed services. We have already experimented with
the use of events and rules for the integration of distributed services in (Jin et al., 2006; Jin et al., 2007).

REFERENCES


ABOUT THE AUTHORS

Yang Xiao received the Ph.D degree in computer science from the Arizona State University in 2006. She is currently a software testing engineer at Microsoft, focusing on integrated development environment testing methodologies and practices. Her research interests include process failure recovery and application-dependent correctness in Grid/Web service composition environment.

Susan D. Urban is a professor in the School of Computing and Informatics at Arizona State University. She received the Ph.D. degree in computer science from the University of Louisiana at Lafayette in 1987. She is the co-author of An Advanced Course in Database Systems: Beyond Relational Databases (Upper Saddle River, NJ: Prentice Hall, 2005). Her research interests include Active/Reactive Behavior in Data-Centric Distributed Computing Environments; Event, Rule, and Transaction Processing for Grid/Web Service Composition; Integration of Event and Stream Processing. Dr. Urban is a member of the Association for Computing Machinery, the IEEE Computer Society, and the Phi Kappa Phi Honor Society.