The DeltaGrid Abstract Execution Model: Service Composition and Process Interference Handling

Yang Xiao, Susan D. Urban, and Ning Liao

Department of Computer Science and Engineering
Arizona State University
PO Box 878809 Tempe, AZ, 85287-8809 USA
{yang.xiao, susan.urban}@asu.edu

Abstract. This paper introduces the DeltaGrid abstract execution model as a foundation for building a semantically robust execution environment for concurrent processes executing over Delta-Enabled Grid Services (DEGS). A DEGS is a Grid Service with an enhanced capability to capture incremental data changes, known as deltas, associated with service execution in the context of global processes. The abstract model contains a service composition model that provides multi-level protection against service execution failure, thus maximizing the forward recovery of a process. The model also contains a process recovery model to handle the possible impact caused by failure recovery of a process on other concurrently executing processes using data dependencies derived from a global execution history and using user-defined correctness criteria. This paper presents the abstract execution model and demonstrates its use. We also outline future directions for incorporating application exception handling and build a simulation framework for the DeltaGrid system.

1 Introduction

In a service-based architecture [17], the correctness of processes composed of distributed Web/Grid services is a concern due to the challenges introduced by the loosely coupled, autonomous, and heterogeneous nature of the execution environment. Compositional serializability [16] is too strong of a correctness criterion for concurrent processes since individual service invocations are autonomous and commit before the process completes. As a result, process execution does not ensure isolation of the data items accessed by individual services, allowing dirty reads and writes to occur.

From an application point of view, dirty reads and writes do not necessarily indicate an incorrect execution, and a relaxed form of correctness dependent on application semantics can produce better throughput and performance. User-defined correctness of a process can be specified as in related work with advanced transaction models [5] and transactional workflows [20], using concepts such as compensation to semantically undo a process. But even when one process determines that it needs to execute compensating procedures, data changes introduced by compensation 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*

D.W. Embley, A. Olivé, and S. Ram (Eds.): ER 2006, LNCS 4215, pp. 40 – 53, 2006.

interference. A robust service composition model should recover a failed process and effectively handle process interference based on application semantics.

This research is defining an abstract execution model for establishing user-defined correctness in a service composition environment. The research is conducted in the context of the DeltaGrid project, which focuses on building a semantically-robust execution environment for processes that execute over Grid Services. The abstract execution model, however, is general enough for use in a Web Service composition environment. Distributed services in the DeltaGrid, referred to as *Delta-Enabled Grid Services (DEGS)* [4], are extended with the capability of recording incremental data changes, known as deltas. Deltas provide the basis for backward recovery of an operation (DE-rollback) and tracking data dependencies among concurrent processes.

The focus of this paper is on the specification of the DeltaGrid abstract execution model, which is composed of a service composition model and a process recovery model. The service composition model provides multi-level protection against service execution failure and maximizes the success of a process execution using compensation, DE-rollback, and contingency. The process recovery model defines a global execution history for distributed process execution, based on which read and write dependencies can be analyzed to evaluate the applicability of DE-rollback and process interference.

The rest of this paper is organized as follows. After outlining related work in Section 2, the paper provides an overview of the DeltaGrid system in Section 3 where the DeltaGrid abstract model has been applied. Section 4 gives an overview of the DeltaGrid abstract execution model. Section 5 presents the service composition model and Section 6 presents the process recovery model. The paper concludes in Section 7 with a summary and discussion of future research.

2 Related Work

Advanced transaction models have been studied to support higher concurrency for long running transactions composed of subtransactions. Sagas [7] can be backward recovered by compensating each task in reverse order. The flexible transaction model [22] executes an alternative path when the original path fails. The backward recovery of a failed transaction causes cascaded rollback or compensation of other transactions that are read or write dependent on the failed transaction. The recent work in [12] removes transactions that are dependent on tainted data produced by a flawed transaction, using multi-version data to track read and write dependencies in a database system. These models cannot well support a service composition environment where dirty reads and writes do not necessarily indicate incorrect execution. As a result, application-dependent correctness criteria should be used.

Research projects in the transactional workflow area have adopted compensation as a backward recovery technique [6, 8, 19] and explored the handling of data dependencies among workflows [8, 19]. The ConTract model [19] compensates a process when a step failure occurs using an approach similar to sagas, and handles data dependencies through pre-/post- condition specification integrated into a workflow script. Forward recovery is not supported in ConTract. WAMO [6] supports backward and forward recovery, but process interference is not considered. CREW [8] requires

a static specification on the equivalence of data items across workflows to track data dependencies. Our research maximizes the forward recovery of a process by a multi-level specification of contingency and compensation. More importantly, we build a global execution history based on which process data dependencies can be analyzed to support application-dependent correctness criteria for handling process interference.

Currently most exception handling work in service composition environments focuses on transaction model implementation and the use of active rules. Open nested transactions over Web Services are supported in [14], contingency is applied to forward recover a composite service in [18], and WS-Transaction [2] defines processes as either Atomic Transactions with ACID properties or Business Activities with compensation capabilities. Rule-based approaches are used to handle service exceptions independent of application logic, such as service availability, selection, and enactment [15, 22], or search for substitute services when an application exception occurs [11]. Our research is among the first to address process interference caused by backward recovery of a service execution failure, and establishes user-defined correctness criteria based on data dependency tracking in a service composition environment.

3 Overview of the DeltaGrid System

The DeltaGrid system, focusing on the semantically robust execution of composite services, provides an execution environment and test bed for the abstract execution model described in this paper. As the fundamental building block, a Delta-Enabled Grid Service (DEGS) is a Grid Service that has been enhanced with an interface to access the deltas that are associated with service execution in the context of globally executing processes [4]. Deltas can be used to undo the effect of a service execution through *Delta-Enabled rollback* (*DE-rollback*). Deltas, in the context of the data dependencies captured in the global process history, can also be used to analyze process interference, determining the effect that the failure recovery of one process can have on other concurrently executing processes.

The GridPML [13] is a process modeling language for the composition of Grid Services in the DeltaGrid system. The GridPML is an XML-based language that supports basic control flow constructs adopted from Web Service composition languages such as BPEL [3] and BPML [1] with features for invoking Grid Services [21]. The GridPML is used in our research to experiment with process execution history capture and has also partially implemented the service composition model [10].

4 The DeltaGrid Abstract Execution Model

The DeltaGrid abstract execution model is composed of 1) the service composition model that defines the hierarchical service composition structure and entity execution semantics, and 2) the process recovery model that tracks process data dependency and handles process interference through active rules.

Table 1 defines the execution entities of the service composition model, with the hierarchical entity composition structure presented in Fig. 1. Operation represents a service invocation and is the basic entity in the composition structure. Compensation is

an operation intended for backward recovery and contingency is an operation for forward recovery. An atomic group contains an operation, an optional compensation, and an optional contingency. A composite group may contain multiple atomic groups or composite groups that execute sequentially or in parallel. A composite group can have its own compensation and contingency as optional elements. A process is defined to be a top-level composite group. The only execution entity not shown in Fig. 1 is the DE-rollback entity. DE-rollback is a system-initiated operation that uses the deltas to reverse an operation execution.

Table 1. Execution Entities

Entity Name	**Definition**
Operation	A DEGS service invocation, denoted as op_{ij}
Compensation	An operation that is used to undo the effect of a commited operation, denoted as cop_{ij}
Contingency	An operation that is used as an alternative of a failed operation (op_{ij}), denoted as top_{ij}
DE-rollback	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 dop_{ij}
Atomic Group	An execution entity that is composed of a primary operation (op_{ij}), an optional compensation (cop_{ij}), and an optional contingency operation (top_{ij}), denoted as $ag_{ij} = <op_{ij} [, cop_{ij}] [,top_{ij}]>$
Composite Group	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 $cg_{ik} = <ag_{ij}^{+} [,cg_{il}^{+}] [,cop_{ik}] [,top_{ik}])>$
Process	A top level composite group

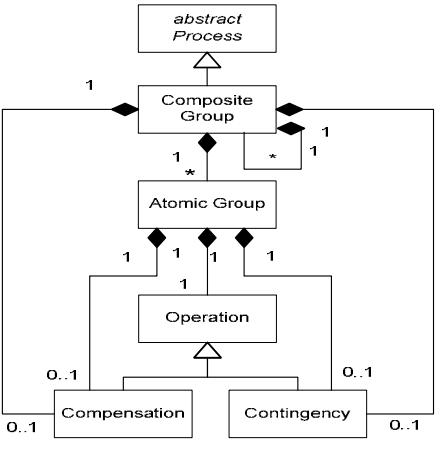

Fig. 1. The Service Composition Structure

Fig. 2 shows an abstract view of a sample process definition. A process p_1 is the top level composite group cg_0. p_1 is composed of two composite groups cg_1 and cg_2, and an atomic group ag_6. Similarly, cg_1 and cg_2 are composite groups that contain atomic groups. Each atomic/composite group can have an optional compensation plan and/or contingency plan, e.g. cg_1 has compensation $cg_1.cop$ and contingency $cg_1.top$ operations. Operation execution failure can occur on an operation at any level of nesting.

The purpose of the service composition model is to automatically resolve operation execution failure using compensation, contingency, and DE-rollback at different composition levels. The next section defines the execution semantics of each entity and addresses operation execution failure handling under various execution scenarios.

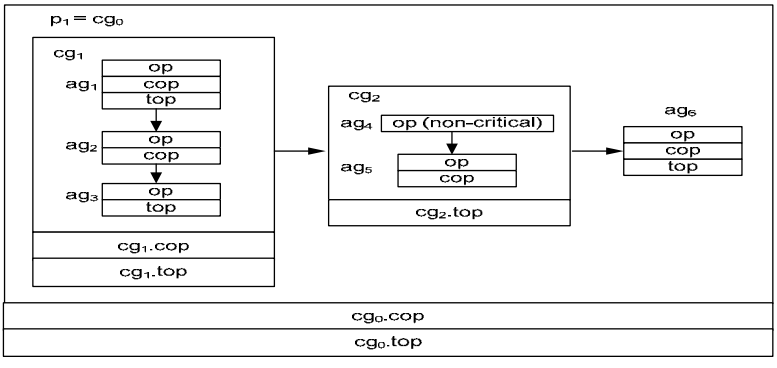

Fig. 2. An Abstract Process Definition

5 Execution Semantics of the Service Composition Model

This section presents the execution semantics of each execution entity and addresses how to resolve operation execution failure. Section 5.1 introduces the DEGS operation execution semantics. Section 5.2 presents the execution semantics of atomic groups. Section 5.3 elaborates on composite group execution semantics.

5.1 DEGS Operation Execution Semantics

Before presenting the execution semantics of a DEGS operation, this section first defines a DEGS operation and its recoverability.

Definition 1. *DEGS operation*. A DEGS operation op_{ij} is a six-element tuple, denoted as $op_{ij} = <I, O, S, R, P, degsID>$, where I is the set of inputs, O is the set of outputs, S is the execution state, R is the pre-commit recoverability, P is the post-commit recoverability, and degsID is a unique identifier of the DEGS where op_{ij} is executed.

Since a DEGS is an autonomous entity, the DeltaGrid system assumes a DEGS guarantees its correctness through proper concurrency control, exposing serializablity or another functionally equivalent correctness criterion to the service composition environment. A DEGS can provide different transaction semantics, supporting an operation as an *atomic* or *non-atomic* execution unit. If a runtime failure occurs, an atomic

operation can automatically roll back. A non-atomic operation stays in a failed state since a service provider is incapable of performing rollback. With the delta capture capability, a DEGS can reverse the effect of an operation through DE-rollback. An operation can have different backward recovery capabilities depending on when the operation needs to be recovered.

Definition 2. *Pre-commit recoverability*. Pre-commit recoverability specifies how an operation can be recovered when an execution failure occurs before the operation completes.

Definition 3. *Post-commit recoverability*. Post-commit recoverability specifies how an operation's effect can be (semantically) undone after it successfully terminates.

Table 2 presents an operation's pre-commit and post-commit recoverability options.

Table 2. Operation Recoverability Options

Recoverability Type	Option	Meaning
Pre-commit	Automatic rollback	The failed service execution can be automatically rolled back by a service provider
	DE-rollback	The failed operation can be undone by executing DE-rollback
Post-commit	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.
	Dismissible	A completed operation does not need any cleanup activities

Fig. 3 compares the execution semantics of a regular service operation shown in (a) with a DEGS operation shown in (b). An 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. An atomic operation can automatically roll back to enter an aborted state. An advantage of DEGS is the use of DE-rollback to undo the effect of a failed operation, thus eliminating the failed state as an operation termination state. However, as a post-commit recovery method, DE-rollback can only be executed under valid process interleaving situations defined as semantic conditions for DE-rollback in Section 6.1.

Definition 4. *Delta*. A delta is a six-element tuple, denoted as Δ(oID, a, V_{old}, V_{new}, ts_n, op_{ij}), representing an incremental value change on an attribute of an object generated by execution of a DEGS operation. A delta contains an object identifier (oID) indicating the changed object, an attribute name (a) indicating the changed attribute, the old value of the attribute (V_{old}) before the execution of the operation, the new value of the attribute (V_{new}) created by the operation, a timestamp (ts_n), and the identifier of the operation (op_{ij}) that has created this delta.

Definition 5. *Runtime context*. The runtime context of an operation is a five-element tuple, denoted as $r(op_{ij}) = <ts_s, ts_e, I, O, S>$. The runtime context of op_{ij} contains a start time (ts_s), end time (ts_e), input (I), output (O), and state (S). Similarly, the runtime context of a process $r(p_i) = <ts_s, ts_e, I, O, S>$.

Definition 6. *Operation execution history*. An operation execution history $H(op_{ij})$ is a four-element tuple, denoted as $H(op_{ij}) = <ts_s, ts_e, \delta op_{ij}, r(op_{ij})>$. ts_s and ts_e are the start and end time of op_{ij}'s execution. δop_{ij} a time-ordered sequence of deltas that are generated by execution of op_{ij}, denoted as $\delta op_{ij} = [\Delta(oID_A, a, V_{old}, V_{new}, ts_1, op_{ij}), \ldots, \Delta(oID_B, b, V_{old}, V_{new}, ts_x, op_{ij}), \ldots, \Delta(oID_D, d, V_{old}, V_{new}, ts_n, op_{ij})]$ $(ts_s < ts_1 < ts_x < ts_n < ts_e)$. $r(op_{ij})$ is the runtime context of op_{ij}.

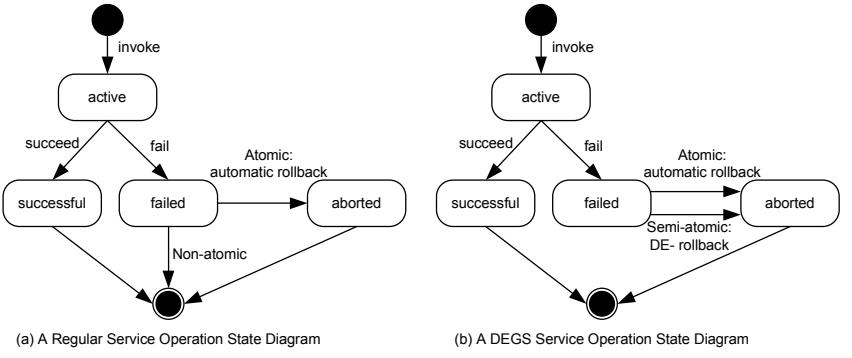

Fig. 3. DEGS Operation Execution Semantics

5.2 Atomic Group (AG) 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 post-commit recovery activity such as DE-rollback or compensation.

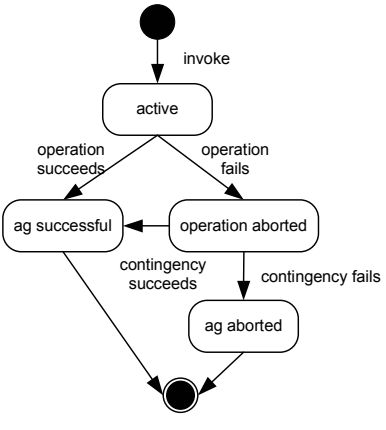

Fig. 4. Atomic Group Execution Semantics

Fig.4 describes the execution semantics of an ag. An ag has four states: {active, operation aborted, ag successful, ag aborted}. The termination states are {ag successful, ag aborted}. An ag enters the active state if the primary operation is invoked. If the primary operation successfully terminates, the ag enters the ag successful state. Otherwise the ag enters the operation aborted state, where contingency will be executed. If the contingency succeeds, the ag enters the ag successful state. Otherwise, the contingency itself is aborted, which leads the ag to the ag aborted state. Compensation and DE-rollback as post-commit recovery techniques for an atomic group are addressed in the context of a composite group execution in the next section.

5.3 Composite Group (CG) Execution Semantics

Before presenting the execution semantics of a composite group, this section first introduces concepts that are related to a composite group. This research has extended shallow/deep compensation originally defined in [9] to be used for a composite group.

Definition 7. *Critical.* An atomic/composite group is critical if its successful execution is mandatory for the enclosing composite group. The execution failure of a *non-critical* group will not impact the state of the enclosing composite group, and the composite group can continue execution. When an execution failure occurs, contingency must be executed for critical groups. Contingency is not necessary for a non-critical group. A group is critical by default.

In Fig.2, if ag_4 fails, cg_2 will continue its execution with ag_5 since ag_4 is non-critical.

Definition 8. *Shallow compensation.* Shallow compensation of a composite group invokes the composite group's compensation.

Definition 9. *Deep compensation.* Deep compensation of a composition group invokes post-commit recovery activity (DE-rollback or compensation) of each critical subgroup.

Shallow compensation is invoked when a composite group successfully terminates but needs a semantic undo due to another operation's execution failure. For example, in Fig. 2, if ag_6 fails, shallow compensation of cg_1 will be executed. When shallow compensation is not specified, a deep compensation is performed. For example, cg_2's deep compensation will be invoked when ag_6 fails since cg_2 does not have shallow compensation. A deep compensation can also be performed for a composite group cg when cg has a subgroup failure before cg completes. The service composition model currently assumes a compensation always succeeds. The model will be extended to handle compensation failure by provision of system-enforced recovery action.

The state of a composite group (cg) is determined by the combined state of the composing subgroups (sg_i), which are either atomic groups or other composite groups. A cg has states: {active, cg successful, sg_i aborted, cg aborted, cg deep compensated}. To simplify the state diagram, cg extended abort is introduced to refer to cg aborted or cg deep compensated. The termination states are: {cg successful, cg extended abort}.

Fig. 5(a) presents the execution semantics of a composite group composed of only atomic groups. A cg remains active during a subgroup's execution. If all the subgroups terminate successfully, a cg enters the cg successful state. If a subgroup ag_i fails, the cg enters the ag_i aborted state. If ag_i is the first subgroup of cg, cg enters the cg aborted

state. Otherwise all of the previously executed subgroups ($ag_{1..i-1}$) will be post-commit recovered, leading cg to the cg deep compensated state. From the cg extended abort state, cg's contingency can be executed to enter the cg successful state. If cg's contingency fails, cg remains in the cg extended abort state.

Fig. 5(b) presents the execution semantics of a composite group (cg) that is composed of subgroups sg_i that can be either atomic groups or composite groups. If any subgroup sg_i fails, sg_i enters the sg_i extended abort state, according to the state transition described in Fig. 4 (if sg_i is an atomic subgroup) and in Fig. 5(a) (if sg_i is a composite subgroup). Other state transitions are the same as defined in Fig. 5(a).

In Fig.2, if ag_5 fails, cg_2's contingency gets executed since ag_4 is non-critical. If ag_6 fails, cg_2 is deep compensated by executing ag_5's compensation since ag_4 is non-critical and needs no compensation. cg_1 is shallow compensated by executing cg_1.cop.

6 The Process Recovery Model

The service composition model has elaborated on how backward recovery (DE-rollback/compensation) and forward recovery (contingency) are applied at different composition levels to maximize the successful execution of a process. After a backward recovery, the data changes introduced by backward recovery of a failed process p_f *potentially* cause a read dependent process p_r or a write dependent process p_w, to be recovered accordingly. Under certain semantic conditions, however, processes such as p_r and p_w may be able to continue running. The process recovery model addresses the applicability of DE-rollback and the handling of process interference based on data dependencies extracted from the global execution history through active rules.

This section presents the process recovery model. Section 6.1 defines process execution history and read/write dependency. Section 6.1 also presents the semantic

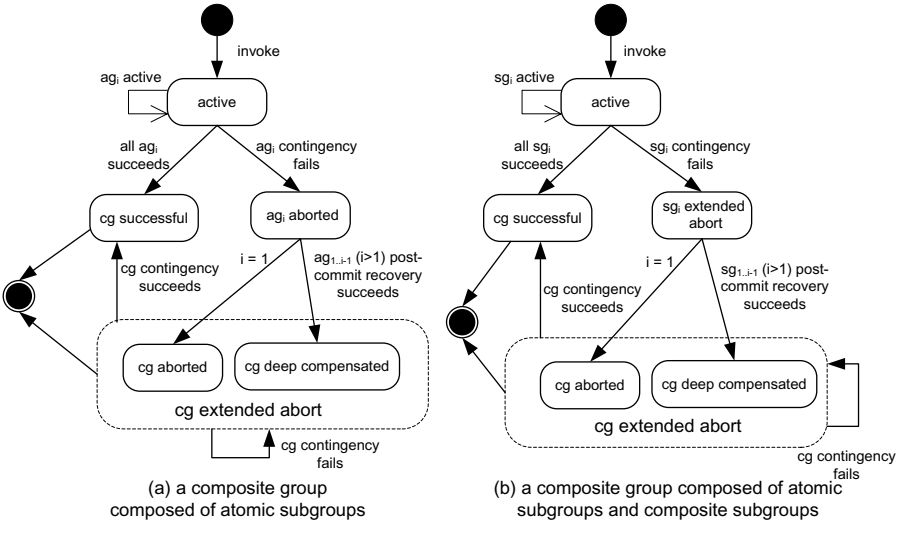

Fig. 5. Composite Group Execution Semantics

condition for DE-rollback. Section 6.2 presents process interference rule specification, and demonstrates its use through an online shopping application.

6.1 Global Execution History and Process Dependency

The global execution history is the foundation for analyzing data dependencies.

Definition 10. *Global execution history*. A global execution history GH is an integration of individual operation execution histories within a time frame, denoted as GH = <ts_s, ts_e, δg, gr>, where:

- ts_s and ts_e form the time frame of the GH.
- δg is a time-ordered sequence of deltas generated by distributed operation execution, denoted as $\delta g = [\Delta(oID_A, a, V_{old}, V_{new}, ts_1, op_{ij}),\ldots, \Delta(oID_B, b, V_{old}, V_{new}, ts_x, op_{kl}),\ldots, \Delta(oID_D, d, V_{old}, V_{new}, ts_n, op_{wz})](ts_s < ts_1 < ts_x < ts_n < ts_e)$.
- gr is the global execution context, which is a time-ordered sequence of runtime context information for operations and processes that occur within the time frame of the global execution history. The global execution context is denoted as $gr = [r(en)|(en=op_{ij}$ or $en=p_i)$ and $(ts_s< r(en).ts_s< r(en).ts_e< ts_e)]$, where en represents an execution entity (either an operation op_{ij} or a process p_i). From gr, we can get a time-ordered sequence of system invocation events, denoted as $E = [e(op_{ij}), \ldots, e(op_{wk}), \ldots]$. E can be used to identify potential read dependencies among processes.

Data dependencies are defined based on information captured in the GH.

Definition 11. *Write dependency*. A write dependency exists if a process p_i, writes a data item *x* that has been written by another process p_j before p_j completes ($i \neq j$).

Definition 12. *Process-level* w*rite dependent set.* A process p_j's write dependent set is a set of all the processes that are write dependent on p_j, denoted as $wd_p(p_j)$.

Definition 13. *Operation-level write dependency*. An operation-level write dependency exists if an operation op_{ik} of process p_i writes data that has been written by another operation op_{jl} of process p_j. An operation-level write dependency can exist between two operations within the same process ($i = j$).

Definition 14. *Operation-level* w*rite dependent set.* An operation op_{jl}'s operational-level write dependent set is a set of all the operations that are write dependent on op_{jl}, denoted as $wd_{op}(op_{jl})$.

Definition 15. *Read dependency*. A *read dependency* exists if a process p_i, reads a data item *x* that has been written by another process p_j before p_j completes ($i \neq j$).

Definition 16. *Read dependent set.* A process p_j's read dependent set contains all the processes that are read dependent on p_j, denoted as $rd_p(p_j)$.

Write dependency can be analyzed from the global execution history GH. Suppose two operations have modified the same data item A. In GH, we observe $\delta g = [\ldots, \Delta(oID_A, a, V_{old}, V_{new}, ts_1, op_{ij}), \ldots, \Delta(oID_A, a, V_{old}, V_{new}, ts_x, op_{kl}), \ldots]$ $(ts_1 < ts_x)$. δg indicates that at operation level, op_{kl} is write dependent on op_{ij}, denoted as $op_{kl} \in wd_{op}(op_{ij})$. At process level, if $k \neq i$, p_k is write dependent on p_i, denoted as $p_k \in wd_p(p_i)$.

Definition 17. *DE-rollback*. DE-rollback of an operation op_{ik} is the action of undoing the effect of op_{ik} by reversing the data values that have been modified by op_{ik} to their before images, denoted as dop_{ik}.

Due to the existence of write dependency, the semantic condition of op_{ik}'s DE-rollback is: a) op_{ik}'s write dependent set is empty $wd_{op}(op_{ik}) = \Phi$, or b) op_{ik}'s write dependent set contains only operations from the same process $wd_{op}(op_{ik}) = \{op_{il}\}$ and the DE-rollback condition holds for each op_{il}. The semantic condition conforms to the traditional notion of recoverability where dirty reads and dirty writes are not allowed.

GH can also reveal potential read dependencies through runtime context. An operation op_{ik} is potentially read dependent on another operation op_{jl} if: 1) op_{ik} and op_{jl} execute on the same DEGS, denoted as $op_{ik}.degsID = op_{jl}.degsID$, and 2) op_{ik} starts after op_{jl}'s invocation, denoted as $r(op_{ik}).ts_s >= r(op_{jl}).ts_s$, or op_{ik} starts before op_{jl}'s invocation and ends after op_{jl}'s invocation, denoted as $r(op_{ik}).ts_s < r(op_{jl}).ts_s$ and $r(op_{ik}).ts_e > r(op_{jl}).ts_s$.

Assume op_{ij} is compensated. In GH, we observe an event sequence $E = [\ldots, e(op_{ij}), e(op_{km}), e(op_{xy}), e(cop_{ij})]$. E shows that op_{km} is invoked after op_{ij}, thus op_{km} is potentially read dependent on op_{ij}. At process level, if $k \neq i$ and p_k is active, p_k is potentially read dependent on p_i. A terminated process is not considered for process interference since a completed process should not be affected by ongoing changes.

Write dependencies and potential read dependencies are resolved through the use of process interference rules as defined in the next subsection.

6.3 Process Interference Rules

Process interference rules are active rules specifying how data change caused by of a process (DE-rollback or compensation) can possibly affect other active processes. This section outlines our initial phase of process interference rule specification which will be formally defined as part of future work. Fig. 6 shows that a process interference rule contains four parts: event, define, condition and action.

Event:	backward recovery event & event filter
Define:	variable declaration
Condition:	process interference evaluation
Action:	recovery command list

Fig. 6. Process Interference Rule Specification

Event contains a backward recovery event and an event filter. A backward recovery event is a DE-rollback $e(dop_{ij})$ or a compensation of an operation $e(cop_{ij})$ or process $e(cp_i)$. Compensation of a process cp_i involves compensation or DE-rollback of the process's operations. The event filter retrieves write and potential read dependent processes. Define declares variables to support condition evaluation and action invocation using the global execution history interfaces. Condition is a Boolean expression evaluating process interference based on application semantics. Action is a list of recovery commands, including backward recovery of a process (bkRecover(p_i)), and re-execution of an operation (re-execute(op_{ij})) or a process (re-execute(p_i)).

The global execution history exposes three types of interfaces: 1) $wd_p(p_i$, dependentProcessName) returns a set of active process instances of a given name (dependentProcessName) that are write dependent on p_i, 2) $rd_p(p_i$, dependentProcessName) returns a set of active process instances of given name (dependentProcessName) that are potentially read dependent on p_i, and 3) $\delta(p_i$, className) returns deltas with a given class name (className) that are generated by p_i.

The rest of the section demonstrates the use of process interference rules using an online shopping application with processes composed of DEGSs. The process placeClientOrder places client orders, decreases the inventory quantity, and possibly increases a backorder quantity. The process replenishInventory increases the inventory quantity when vendor orders are received and possibly decreases the backorder quantity. Several process instances could be running at the same time.

Write dependency scenario: Failure of a replenishInventory process could cause a write dependent placeClientOrder process to be compensated (since the items ordered may not actually be available). However, compensation of a placeClientOrder process would not affect a write dependent replenishInventory or returnClientOrder process.

Fig. 7 shows a process interference rule specifying that if a replenishInventory process (p_{ri}) is compensated, a placeClientOrder process with an inventory item that has been supplied by p_{ri} must be compensated and re-executed. In Fig. 7:

- $e(cp_{ri})$ represents the compensation of process replenishInventory.
- $wd_p(p_{ri}$, "placeClientOrder") returns a set of instances of process placeClientOrder that are write dependent on process p_{ri}. The event filter $wd_p(p_{ri}$, "placeClientOrder") $\neq \Phi$ verifies the existence of a placeClientOrder process that is write dependent on p_{ri}.
- Process p_w declares p_w as a process instance. $p_w \in wd_p(p_{ri}$, "placeClientOrder") restricts that p_w must be a placeClientOrder process that is write dependent on p_{ri}.
- $\delta(cp_{ri}$, "InventoryItem") retrieves deltas of class InventoryItem that are generated by the compensation of p_{ri}. Similarly, $\delta(p_w$, "InventoryItem") retrieves deltas of class InventoryItem that are generated by p_w. Condition $\delta(cp_{ri}$, "InventoryItem") $\cap$ $\delta(p_w$, "InventoryItem") $\neq \Phi$ evaluates if the cp_{ri} and p_w process the same inventory item.
- bkRecover(p_w) is a recovery command to backward recover p_w, based on p_w's composition structure and recoverability. re-execute(p_w) is a command to re-execute p_w.

Event:	$e(cp_{ri})$ & $wd_p(p_{ri}$, "placeClientOrder") $\neq \Phi$
Define:	Process p_w ($p_w \in wd_p(p_{ri}$, "placeClientOrder"))
Condition:	$\delta(cp_{ri}$, "InventoryItem") $\cap$ $\delta(p_w$, "InventoryItem") $\neq \Phi$
Action:	bkRecover(p_w);
	re-execute(p_w);

Fig. 7. A Sample Process Interference Rule Handling Write Dependency

As a summary, after recovery of a failed process, process interference must be identified. However write/read dependent processes *may or may not* need to be recovered, depending on the application semantics defined by a process interference rule.

7 Summary and Future Directions

This paper has presented an abstract execution model as the foundation for building a semantically robust execution environment for distributed processes over Delta-Enabled Grid services. We are developing the DeltaGrid system to support the abstract model. We have implemented several major architectural components such as DEGS [4], the GridPML [10, 13], and a process history capture system (PHCS) [21] as a logging mechanism for distributed processes. The PHCS fully implements the global execution history interfaces to evaluate the applicability of DE-rollback and process interference.

This research contributes towards establishing a semantically robust execution model for distributed processes executing over autonomous, heterogeneous resources. A unique aspect of this research is the provision for multi-level protection against service execution failure, and handling process interference based on application-dependent correctness criterion integrated with data dependency tracking through a global execution history.

Our future direction is to provide a complete support of application-dependent correctness criterion by refining the definition of process interference rules and incorporating application exceptions rules in the process recovery model. We are building a simulation framework for the DeltaGrid system to demonstrate the concepts defined in the abstract model in a distributed service composition environment.

References

1. Business Process Modeling Language, http://www.bpmi.org/specifications.esp, 2002.
2. Specification: Web Services Transaction (WS-Transaction), http://www-106.ibm.com/ developerworks/webservices/library/ws-transpec/, 2002.
3. Specification: Business Process Execution Language for Web Services Version 1.1, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/, 2003.
4. Blake, L., *Design and Implementation of Delta-Enabled Grid Services*, MS Thesis, Dept. of Comp. Sci. and Eng., Arizona State Univ., (2005).
5. de By, R., Klas, W., Veijalainen, J., *Transaction Management Support for Cooperative Applications*. 1998: Kluwer Academic Publishers.
6. Eder, J., Liebhart, W., "The Workflow Activity Model WAMO," Proc. of the *3rd Int. Conference on Cooperative Information Systems (CoopIs)*, 1995.
7. Garcia-Molina, H., Salem, K., "Sagas," Proc. of the *ACM SIGMOD Int. Conference on Management of Data*, 1987.
8. Kamath, M., Ramamritham, K., "Failure Handling and Coordinated Execution of Concurrent Workflows," Proc. of the *IEEE Int. Conference on Data Engineering*, 1998.
9. Laymann, F., "Supporting Business Transactions via Partial Backward Recovery in Workflow Management," Proc. of the *GI-Fachtagung für Datenbanksysteme in Business*, Technologie und Web (BTW'95), 1995.
10. Liao, N., *The Extened GridPML Design and Implementation*, MCS Project Report, Dept. of Comp. Sci. and Eng., Arizona State Univ., (2005).
11. Lin, F., Chang, H., "B2B E-commerce and Enterprise Integration: The Development and Evaluation of Exception Handling Mechanisms for Order Fulfillment Process Based on BPEL4WS," Proc. of the *7th IEEE Int. Conference on Electronic commerce*, 2005.

12. Lomet, D., Vagena, Z., Barga, R., "Recovery from "Bad" User Transactions," Proc. of the *ACM SIGMOD Int. Conference on Management of Data*, 2006.
13. Ma, H., Urban, S. D., Xiao, Y., and Dietrich, S. W., "GridPML: A Process Modeling Language and Process History Capture System for Grid Service Composition," Proc. of the *IEEE Int. Conference on e-Business Engineering*, 2005.
14. Mikalsen, T., Tai, S., Rouvellou, I., "Transactional Attitudes: Reliable Composition of Autonomous Web Services," Proc. of the *Workshop on Dependable Middleware-based Systems (WDMS)*, part of the *Int. Conference on Dependable Systems and Networks (DSN)*, 2002.
15. Shi, Y., Zhang, L., Shi. B., "Exception Handling of workflow for Web services," Proc. of the *4th Int. Conference on Computer and Information Technology*, 2004.
16. Singh, M.P., Huhns, M. N., *Service-Oriented Computing*. 2005: Wiley.
17. Business Service Grid: Manage Web Services and Grid Services with Service Domain Technology, http://www-128.ibm.com/developerworks/ibm/library/gr-servicegrid/, 2003.
18. Tartanoglu, F., Issarny, V., Romanovsky, A., Levy, N., "Dependability in the Web Services Architecture," Proc. of the *Architecting Dependable Systems, LNCS 2677*, 2003.
19. Wachter, H., Reuter, A., "The ConTract Model," in *Database Transaction Models for Advanced Applications*, A. Elmagarmid, Editor. 1992.
20. Worah, D., Sheth, A., "Transactions in Transactional Workflows," in *Advanced Transaction Models and Architectures*, S. Jajodia, and Kershberg,L., Editor, Springer.
21. Xiao, Y., Urban, S. D., Dietrich, S., "A Process History Capture System for Analysis of Data Dependencies in Concurrent Process Execution," Proc. of the *2nd Int. Workshop on Data Engineering Issues in E-Commerce and Services*, 2006.
22. Zeng, L., Lei, H., Jeng, J., Chung, J., Benatallah, B., "Policy-Driven Exception-Management for Composite Web Services," Proc. of the *7th IEEE Int. Conference on E-Commerce Technology*, 2005.