A Process History Capture System for Analysis of Data Dependencies in Concurrent Process Execution

Yang Xiao[1], Susan D. Urban[1], and Suzanne W. Dietrich[2]

[1] Department of Computer Science and Engineering
Arizona State University
PO Box 878809 Tempe, AZ, 85287-8809 USA
[2] Department of Mathematical Sciences and Applied Computing,
Arizona State University
PO Box 37100, Phoenix, AZ, 85069-7100 USA
{yang.xiao, susan.urban, dietrich}@asu.edu

Abstract. This paper presents a Process History Capture System (PHCS) as a logging mechanism for distributed long running business processes executing over Delta-Enabled Grid Services (DEGS). A DEGS is a Grid Service with an enhanced interface to access incremental data changes, known as deltas, associated with service execution in the context of global processes. The PHCS captures process execution context and deltas from distributed DEGSs and constructs a global schedule for multiple executing processes, integrating local schedules that are extracted from deltas at distributed sites. The global schedule forms the basis for analyzing data dependencies among concurrently executing processes. The schedule can be used for rollback and also to identify data dependencies that affect the possible recovery of other concurrent processes. This paper presents the design of the PHCS and the use of the PHCS for process failure recovery. We also outline future directions for specification of user-defined semantic correctness.

1 Introduction

There is an increasing demand for integrating business services provided by different service vendors to achieve collaborative work in a distributed environment. With the adoption of Web Services and Grid Services [9], many of these collaborative activities are long-running processes based on loosely-coupled, multi-platform, service-based architectures [20]. These distributed processes pose new challenges for execution environments, especially for the semantic correctness of concurrent process execution. The concept of serializability is too strong of a correctness criterion for concurrent distributed 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 of the process, allowing dirty reads and dirty writes to occur. User-defined correctness of a process can be specified as in related work with advanced transaction models [6, 8] and transactional workflows [25], using concepts such as compensation to semantically undo a process. But even when one process determines that it needs to execute compensating procedures, the affect of the compensation on concurrently executing processes is

J. Lee et al. (Eds.): DEECS 2006, LNCS 4055, pp. 152 – 166, 2006.

difficult to determine since there is no knowledge of data dependencies among concurrently executing processes. Data dependencies are needed to determine how the data changes caused by the recovery of one process can possibly affect other processes that have read or written data modified by the failed process.

This research is investigating a process history capture system that records the execution history of distributed processes that execute over Grid Services. The research is being conducted in the context of the DeltaGrid project, which is focusing on the development of a semantically-robust execution environment for the composition of Grid Services. Distributed services in the DeltaGrid environment are extended with the capability of recording incremental data changes, known as *deltas*. These services are referred to as *Delta-Enabled Grid Services (DEGS)* [5]. The deltas generated by DEGS are forwarded to a process history capture system that organizes deltas from distributed sources into a time-sequenced schedule of data changes. Deltas can then be used to undo the effect of a service execution. This undo process is referred to as *Delta-Enabled rollback* (*DE-rollback*). Deltas can also be used to analyze data dependencies among concurrently executing processes to determine how the failure and recovery of one process can potentially affect other data-dependent processes. We refer to this situation as *process interference* and are investigating how user-defined process interference rules can be used to specify how a read or write dependent process reacts to data changes caused by the failure recovery of a process on which it depends.

The focus of this paper is on the design of the process history capture system, with an illustration of how deltas are organized into a schedule that can be analyzed to reveal data dependencies. The unique aspect of the process execution history capture system is that it provides a basis for analyzing the effects that failure or recovery techniques for one process may have on other concurrently executing processes. The analysis of the process history can be used to support the recovery process at run-time or to support the testing of distributed processes, providing a means to evaluate recovery plans, or to discover the need for the specification of recovery techniques under different circumstances.

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, including the system architecture and the design of Delta-Enabled Grid Services. The design of the process history capture system is presented in Section 4, with a focus on the indexing structure for constructing the time-ordered sequence of deltas. Section 5 then illustrates how the delta schedule can be used to support rollback and analysis of dependencies, with scenarios provided from an online shopping application. The paper concludes in Section 6 with a summary and discussion of future research.

2 Related Work

Research projects in the transactional workflow area have adopted compensation as a backward recovery technique [7, 12, 23] and explored the handling of data dependencies among workflows [12, 23]. The ConTract model [23] considers the data changes caused by a workflow execution or compensation on other workflows by specifying pre- and post- condition for a step in a workflow. Exception handling is integrated

with the normal flow of control, making it difficult to determine if a data change is caused by an exception handling action or the normal workflow execution. METEOR [24] handles pre-defined exceptions in a Java try-catch fashion within workflow specification. METEOR uses a hierarchical error model based on which errors can be handled at the task level, the task manager level, or the workflow engine level. In CREW [12], a static specification must be given about which data item in one workflow is equivalent to a data item in another workflow to track the dependencies among workflows. The research in transactional workflows has explored application exception handling using an embedded approach. However there is no a satisfactory solution to track process dependencies and a flexible handling mechanism separated from normal workflow control flow for the effect of failure recovery. Our research provides a logging mechanism for distributed processes execution based on which process dependencies are analyzed to support the evaluation and handling of impact of a failure recovery activity using active rules.

Exception handling in current service composition environments has been addressed through the specification of transaction semantics and through the use of rules. Transactional Attitudes (TA) [16] supports the open nested transaction over Web Services. TA provides interfaces where a service provider declares a service's transaction capabilities, and clients express transaction requirement. Services satisfying a client's requirements can form a transaction with required semantics. WS-Transactions [1] supports processes composed of Web Services as either Atomic Transactions with ACID properties or Business Activities with compensation capabilities. Web Service Composition Action (WSCA) [21] supports contingency as a forward recovery mechanism of a composite service. Rule-based approaches are also used to handle service exceptions independent of application logic, such as service availability, selection, and enactment [18, 26]. The work in [14] has experimented with providing forward recovery for a process by searching for substitute services when an application exception occurs. How to flexibly specify the handling of application exceptions and the impact of failure on other processes has not been addressed in a service composition environment. Our research is among the first to address process interference caused by backward recovery of a failed process in a service composition environment. Specifically, our research records execution history of concurrently executing processes over distributed services. The process execution history forms the basis to analyze process dependency and handle the process interference.

3 Overview of the DeltaGrid System

Before describing the manner in which the process history capture system is used to support recovery in the DeltaGrid environment, this section introduces the architectural components of the DeltaGrid system. Section 3.1 outlines the design of Delta-Enabled Grid Services as the building blocks of the DeltaGrid environment. Section 3.2 presents an abstract view of the Grid Process Modeling Language for the specification of processes. Section 3.3 then presents the architecture of the DeltaGrid system.

3.1 Delta-Enabled Grid Services (DEGS)

A DEGS is a Grid Service that has been enhanced with an interface that provides access to the incremental data changes, known as *deltas*, which are associated with service execution in the context of globally executing processes. Deltas can support DE-rollback as a backward recovery mechanism for an operation and also provide the basis for discovering data dependencies among processes.

The design of DEGS is based on our past research with capturing object deltas [19] in an object-oriented database and defining delta abstractions for the incremental, runtime debugging of active rule execution [4, 5, 22]. Whereas object deltas capture incremental state changes, delta abstractions define views over the object deltas, providing different granularity levels for examining the state changes that have occurred during the execution of active rules and update transactions.

Fig. 1 presents the architecture of the delta capture and storage subsystem of DEGS. A client application invokes operations on a DEGS. A delta event processor receives deltas back from the DEGS, which are used by the process history capture system to maintain a global log of data dependencies.

A DEGS uses an OGSA-DAI Grid Data Service [11] for database interaction. The database 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 [17]. 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 over the source database are stored in a delta repository. Deltas can also 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.

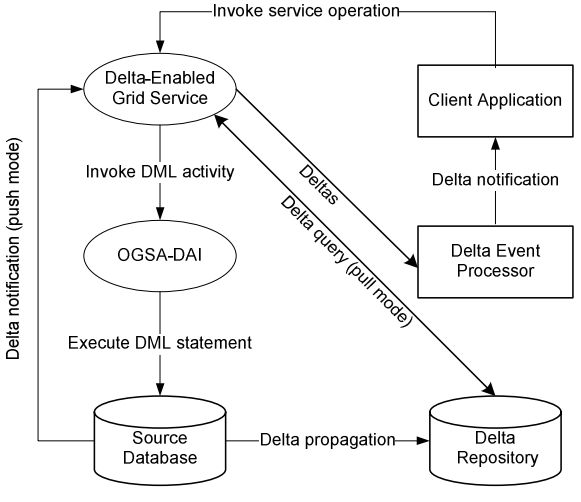

Fig. 1. Delta Capture and Storage Subsystem Architecture [5]

The DEGS uses the object delta concept [19] to create a conceptual view of relational deltas. As shown in Fig. 2, 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) that the associated object belongs to, 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 PropertyValues (i.e., delta values). Each PropertyValue contains the actual delta value and 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 Fig. 2 [5].

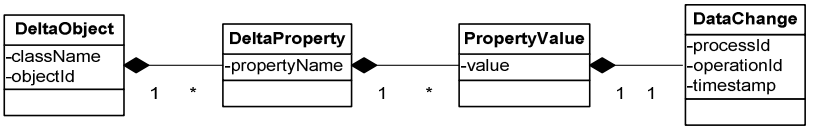

Fig. 2. The Delta Structure in DEGS

3.2 The Grid Process Modeling Language (GridPML)

A process modeling language for the composition of Grid Services, known as the GridPML [15], has been designed to define a process 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 [2] with features for invoking Grid Services. Instead of competing with languages such as BPEL, the GridPML is primarily used as a flexible implementation environment to experiment with process execution history capture in the context of our failure recovery work.

Fig. 3 presents an abstract view of a process defined using the GridPML with semantics similar to sagas [10]. The advanced features of the GridPML, such as supporting composite nested groups of service invocations [13], are not shown in this paper for the purpose of simplicity. As shown in Fig. 3, a process (p_i) contains multiple operations (op_{ij}) representing service invocations, where each operation can have an optional compensation activity (cop_{ij}). A compensation activity is used to semantically undo the effect of an operation. A process completes after the successful execution of each operation. As in the saga model [10], a process can be semantically undone by executing the compensation activity for each operation defined in the original control flow in reverse execution order. With DE-rollback supported as a backward recovery mechanism in the DeltaGrid, a process can also be semantically undone by executing DE-rollback on each executed operation as long as proper data dependency conditions are satisfied. The applicability of DE-rollback is discussed in more detail in Section 5.

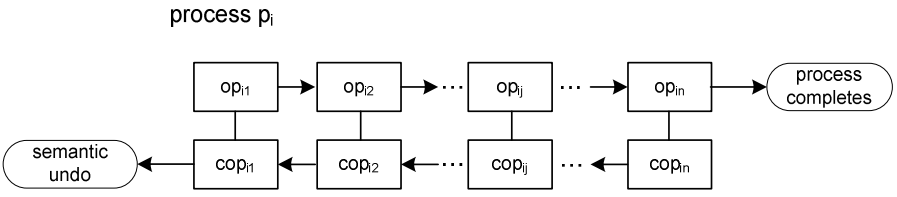

Fig. 3. An Abstract View of GridPML-defined Process

3.3 Architecture of the DeltaGrid System

Fig. 4 presents the architecture of the DeltaGrid system, providing a vision for the DeltaGrid approach to process failure recovery. As a messaging hub, the DeltaGrid event processor receives different types of events and dispatches them to appropriate handlers. Currently three types of events are considered: deltas events, system failure recovery events, and application exception events. Delta events are notifications about the arrival of deltas from DEGS. System recovery events are events that are associated with process failures as well as backward recovery actions such as compensation and DE-rollback. Application exceptions are events that are related to process execution but are outside of the normal flow of execution (i.e., a client cancels an order while the order is in progress).

The processor execution engine parses process scripts, invokes operations on DEGSs, and records process execution context into the process history capture system. The metadata manager stores process and rule metadata. The process metadata contains process schema and scripts described using the GridPML. The rule metadata stores active rules that will be used to handle application exceptions and process failure recovery interference based on user-defined correctness conditions. The rule processor evaluates rule conditions and invokes rule actions.

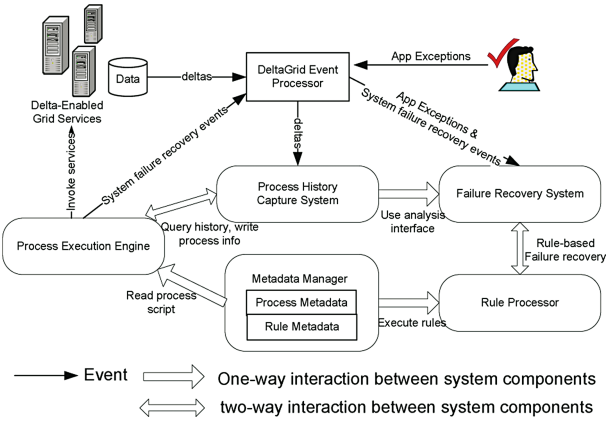

Fig. 4. The DeltaGrid Architecture

The Process History Capture System (PHCS) logs distributed process execution, integrating deltas from DEGSs and process execution context from the execution engine. More importantly, the PHCS provides analysis interfaces to determine data dependencies and evaluate process interference. The failure recovery system generates recovery scripts to resolve application exceptions and associated failure recovery interference. The failure recovery system uses the analysis interfaces provided by the PHCS to generate commands for backward recovery of a failed process and to evaluate if read or write dependent processes need to be recovered based on process interference rules.

We have implemented DEGS, the process execution engine, and the process history capture system. The design and implementation of other system components are in progress. The remainder of this paper describes the PHCS and illustrates the manner in which it is being used in our research to analyze data dependencies.

4 The Process History Capture System (PHCS) Design

This section presents the design of the PHCS. Section 4.1 describes the internal organization of the PHCS. Section 4.2 elaborates on the global delta object schedule that is created to support the analysis of data dependencies for concurrently executing processes.

4.1 Architecture of the PHCS

The PHCS is composed of three layers: the data storage layer, the data access layer, and the service layer, as shown in Fig. 5. The data storage layer stores process execution history, which includes a delta repository and a process runtime information repository. The delta repository stores deltas collected from distributed sites as Java objects organized according to the delta object model presented in Fig. 2. Additional site information is captured to indicate the source site for deltas. The process runtime information repository traces process execution context, such as the identifier, start time, end time, and execution status for each process and operation, input and output variables, and events that are associated with an operation invocation, with details described in [13, 15]. The data storage layer also contains a *global delta object schedule,* which orders DataChange objects from multiple DEGSs according to the time sequence in which they have occurred. Since DataChange objects relate global process execution activities with the delta values that they generate, the global delta object schedule serves as a logging mechanism that reveals write dependencies among processes.

The data access layer provides three components: the deltaAccess interface to read from and write to the delta repository, the processInfoAccess interface to access the process runtime information repository, and the globalScheduleAccess interface for creation and access of the global delta object schedule. The GridPML execution engine updates the process runtime information repository through the processInfoAccess interface.

The service layer receives and parses deltas sent through XML files, and provides the execution history query interface and data dependency analysis interface to other

DeltaGrid system components, such as the failure recovery system and the rule processor. The process history analyzer provides three different capabilities: 1) an execution context interface to access process runtime information, 2) a delta retrieval interface to get deltas, and 3) an analysis interface to derive data dependencies among processes and to determine the applicability of DE-rollback for a process.

Since the global delta object schedule is the primary focus of this paper, the following subsection elaborates on the internal structure of the schedule.

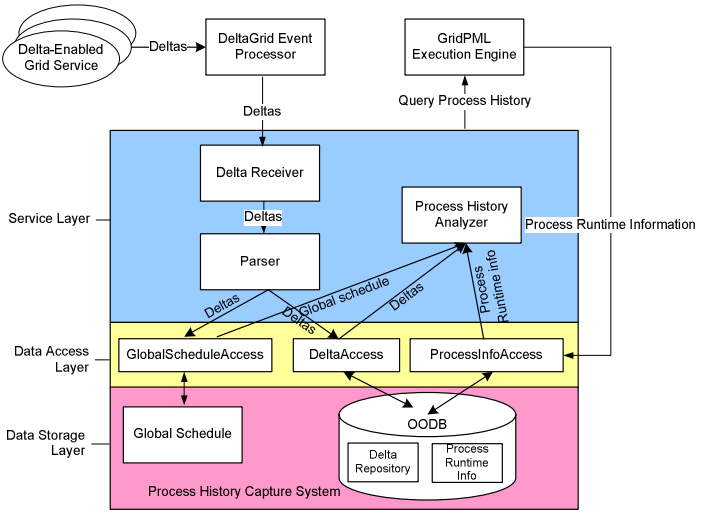

Fig. 5. The PHCS Architecture

4.2 Global Delta Object Schedule

The global delta object schedule forms a time-sequenced log of the delta objects that have been generated by concurrently executing processes. The global schedule is used to discover write dependencies, which can be further used to determine if DE-rollback is applicable as a backward recovery mechanism for a failed process, or if compensation must be invoked. More importantly, it assists in the evaluation of process failure recovery interference and serves as a monitoring mechanism for user-defined correctness.

Fig. 2 illustrates that each DataChange object associated with a delta contains a timestamp, which forms a local schedule for each DeltaProperty. When deltas are received by the PHCS, the DataChange objects provide the basis for ordering deltas. DataChange objects also serve as a link to the delta values stored in the delta repository. To assist online process failure recovery, the global schedule is built as an in-memory data structure that orders DataChange by timestamp, with a two-level index accelerating delta access through process and operation identifiers.

Fig. 6 presents a conceptual view of the global schedule and its relationship to the persistent delta repository and process runtime information repository. A global schedule contains a list of Nodes ordered by timestamp from all the active processes.

Each Node contains a className, objectId, and a propertyName, representing the data modified by a specific operation as identified by a processId and an operationId.

A time-sequence index is built to retrieve a Node through given process and operation identifiers. The index establishes a one-to-one mapping between a Node and a process-operation pair that has made the modification represented by the Node. An entry of the time-sequence index contains a processId, an operationId, and a timestamp. A time-sequence index entry also contains a sequence number (seqNum) to internally differentiate multiple objects with the same timestamp value. The Node and TimeSequenceIndex together serve as a link to the delta repository. Node can be used to access DeltaObject and DeltaProperty instances in the delta repository. The TimeSequenceIndex points to a specific DataChange object. Thus, a Node and TimeSequenceIndex can be used together to access a specific delta value.

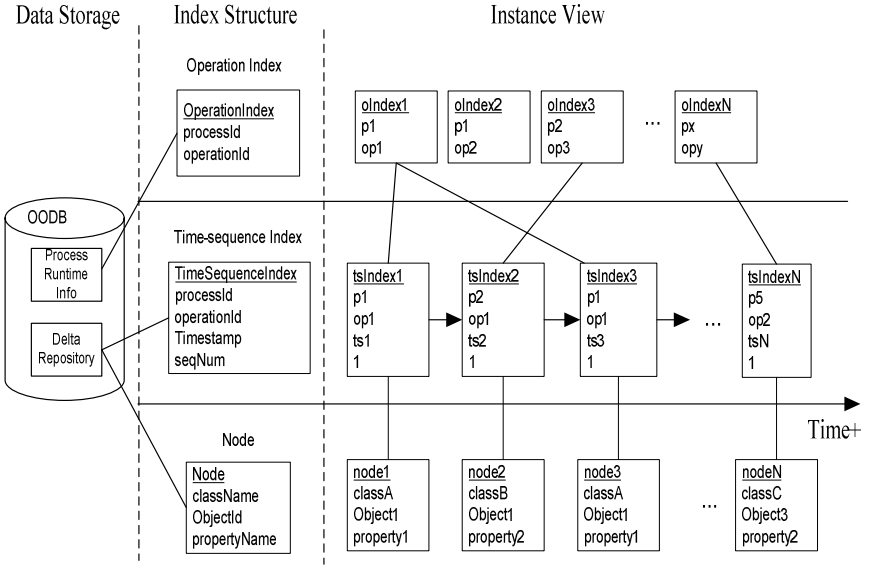

Fig. 6. A Conceptual View of the Global Schedule

When a process fails, the DeltaGrid system queries the global schedule about the processes having write dependencies on the failed process without knowing the details of the timestamp and internal sequence number. To answer this query, an operation index is built on the top of the time-sequence index. An entry of the operation index contains a processId and an operationId. From an operation index entry, all the time-sequence index entries of the same processId and operationId can be retrieved, finding all of the objects that have been modified by this given operation identified by the process-operation pair, and all the interleaved modifications made by other processes during the execution timeframe of the given operation. If only a processId is given, the PHCS can query the process runtime information repository to find all the operations that belong to this specific process. Then the process-operation pair can be used to retrieve information from the global schedule.

5 Use of PHCS

This section illustrates how the PHCS is used to analyze data dependencies. Section 5.1 defines terminology and semantic conditions for the use of DE-rollback. Section 5.2 elaborates on writes dependency scenarios, while Section 5.3 addresses read dependency scenarios.

5.1 Terminology and DE-Rollback Conditions

In a service composition environment, the act of semantically undoing the effect of completed operations within a process is referred to as *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. This situation is called *process failure recovery interference*. Under certain semantic conditions, however, processes such as p_r and p_w may be able to continue running. In the DeltaGrid, we are investigating the use of process interference rules that allow users to flexibly specify how to handle p_r and p_w.

DE-rollback can be used as a backward recovery mechanism under appropriate semantic conditions based on read and write dependencies among concurrently executing processes. In a process execution environment, 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 ($i \neq j$). In this case, p_i is *read dependent* on p_j with respect to x. 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 ($i \neq j$). In this case, p_i is write dependent on p_j with respect to x. To address the applicability of DE-rollback, write dependency must be refined at the operation level.

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 ($i \neq j$) . At the operation level, op_{ik} is write dependent on op_{jl}, thus op_{ik} is write dependent on p_j. At the process level, p_i is write dependent on p_j. When op_{ik} and op_{jl} belong to the same process ($i = j$), op_{ik} is still write dependent on op_{jl}. So operation-level write dependency might exist between two operations within the same process.

DE-rollback can be applied at either the operation level or process level. DE-rollback of an operation op_{ik} is the process of undoing the effect of op_{ik} by reversing the data values that have been modified by op_{ik} to their before images. Since a process is hierarchically composed of operations, DE-rollback of a process p_i, or *full DE-rollback* of p_i, involves the invocation of DE-rollback for every operation in p_i in reverse execution order. A *partial DE-rollback* of a process p_i can be performed if the semantic condition for a full DE-rollback of a process is not satisfied. A partial DE-rollback of a process involves the invocation of DE-rollback for those operations where DE-rollback conditions are satisfied, and compensation for other operations where DE-rollback conditions are not satisfied.

The application of DE-rollback as a backward recovery mechanism must conform to the following semantic conditions:

1) DE-rollback can be performed on an operation op_{ik}, denoted as dop_{ik}, when op_{ik} has no write dependent operations. If op_{ik} is write dependent on another operation within the same process op_{il} (k < l), DE-rollback can be performed on op_{ik} if DE-rollback will be performed on op_{il}. Otherwise compensation should be invoked.

1) A full DE-rollback can be performed on a process only when DE-rollback can be performed on every operation in reverse execution order.

2) If the condition in 2) does not hold, a partial DE-rollback can be performed on a process by invoking DE-rollback of the operations without write dependent operations, and compensation of other operations with write dependent operations. Suppose a process p_i has n executed operations as an ordered list, denoted as $p_i = [op_{i1}, op_{i2}, \ldots, op_{in}]$. This list can be divided into two sets: a *DE-rollback set* denoted as DERBS = $\{op_{ik}\}$ (1<= k <= n) where op_{ik} has no write dependent operations from other processes, and a *compensation set* denoted as COMPS = $\{op_{ij}\}$ (1<= j <= n & j $\neq$ k) where op_{ij} has write dependent operations from other processes. Partial rollback of p_i invokes DE-rollback or compensation on each executed operation of p_i in reverse order of the original execution list, depending on which set an operation belongs to. If an operation is in the DE-rollback set, then DE-rollback will be performed. Otherwise compensation will be performed. Partial DE-rollback also requires that the compensation of an operation which appears later in the original execution list should not affect DE-rollback of an operation that appears earlier in the list. Suppose a process has two operations op_{ik} and op_{ij} (1 <= k < j <= n), op_{ij} requests compensation (cop_{ij}) and op_{ik} requests DE-rollback (dop_{ik}). If cop_{ij} modifies data written by op_{ik}, execution of dop_{ik} will erase the change made by cop_{ij}. So partial DE-rollback of p_i requires cop_{ij} does not modify data that have been written by op_{ik}. Otherwise compensation instead of DE-rollback should be invoked for the earlier operation op_{ik}.

3) After backward recovery (either DE-rollback or compensation), read and write dependent processes should be evaluated for possible recovery actions.

5.2 Write Dependencies

When a process p fails, the DeltaGrid system needs to determine the following information: 1) does p have any write dependent processes, and 2) can DE-rollback be fully or partially applied to p. From the global schedule, a *write dependency table* can be constructed to determine the applicability of DE-rollback on a process. A write dependency table reveals operation level write dependency on a failed process.

Fig. 7 gives a simplified view of a global schedule containing three concurrently executing processes p_1, p_2 and p_3. Each entry has a timestamp (t_x, x = 1, 2, …), a processId (p_i, i = 1, 2, 3), an operationId (op_{ij}, j = 1, 2, …), and an objectId (A,B,…) identifying a modified object.

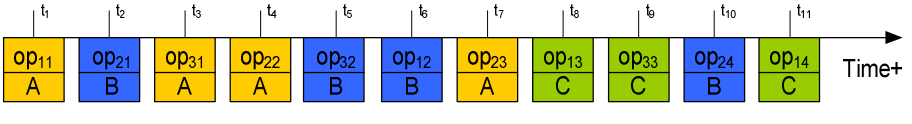

Fig. 7. A Sample Global Delta Object Schedule with Write Dependency on p_1

Suppose process p_1 fails due to execution failure of op_{14}. Table 1 is the write dependency table constructed from the global delta object schedule presented in Fig 7. We can derive the following information from Table 1:

1) p_1 has two write dependent processes: p_2 and p_3.
2) A partial DE-rollback can be performed on p_1. The DE-rollback set of p_1 is $\{op_{14}\}$. The compensation set of p_1 is $\{op_{13}, op_{12}, op_{11}\}$.
3) The backward recovery of p_1 involves the DE-rollback of op_{14} and the compensation of op_{13}, op_{12}, and op_{11}, denoted as a list of failure recovery activities [dop_{14}, cop_{13}, cop_{12}, cop_{11}].

Table 1. Write Dependency Table for p_1

p_1 operations	Objects modified	Dependent operations	Dependent process
op_{14}	C	None	None
op_{13}	C	op_{33}, op_{14}	p_3
op_{12}	B	op_{24}	P_2
op_{11}	A	op_{31}, op_{22}, op_{23}	p_2, p_3

If backward recovery of p_1 causes its write dependent processes p_2 and p_3 to be backward recovered, then a write dependency table for p_2 and p_3 should also be constructed. Backward recovery of a process might cause cascading backward recovery of write dependent processes. We are investigating the development of process interference rules to minimize the recovery of processes according to application logic.

Scenario: Consider an online shopping application with processes that execute over Delta-Enabled Grid services. The process placeClientOrder is responsible for invoking services that place client orders, decrease the inventory quantity, and possibly increase a backorder quantity. The process returnClientOrder is responsible for processing client returns, and invoking services that increase the inventory quantity and possibly decrease the backorder quantity. The process replenishInventory invokes services that increase the inventory quantity when vendor orders are received and possibly decrease the backorder quantity. Furthermore, several instances of each process could be running at the same time. If anyone of these processes fail while an operation is updating the inventory quantity, DE-rollback could be used to restore the inventory value for the failed operation as well as any other completed operations of the process as long as there are no write dependent operations from other processes. If write dependencies exist, the failed process can only be restored through compensation. Furthermore, write dependent processes must be identified and *may or may not* need to be compensated. The need to perform a compensating action depends on the semantics of the situation. For example, 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, failure and compensation of a placeClientOrder process would not affect a write dependent replenishInventory or returnClientOrder process. User-defined process interference rules are needed to define how to respond to such conditions.

5.3 Read Dependencies

The failure of a process can also affect read dependent processes. The delta repository, however, does not capture read activity from DEGS. In this case, the process runtime information component of the PHCS can be queried to identify the active processes that overlap with the execution of failed processes since the runtime information contains the start time of each process. These processes represent *potential* conflicts, but there is no explicit statement of data items that have been read. Techniques are needed to express user-defined process interference rules that define whether or not a process can continue execution or invoke compensating procedures.

Scenario: In the online shopping application, there is a placeVendorOrder process that places orders for inventory items that are below a certain threshold. Suppose the placeVendorOrder process is read dependent on a placeClientOrder process (placeVendorOrder reads the inventory quantity after placeClientOrder writes the inventory quantity). If placeClientOrder fails, placeVendorOrder will be identified as a concurrently executing process. A process interference rule could be expressed to define the conditions under which placeVendorOrder is restarted. If the quantity of the failed placeClientOrder processes is small or if there is no intersection between the items on the vendor order and the client order, then there may be no need to redo placeVendorOrder. If the quantity of the failed placeClientOrder is large, then placeVendorOrder may need to be restarted to avoid overstocking any items that the two processes may have in common. The next phase of our work is addressing how to express such process interference rules, which may require querying the delta repository.

6 Summary and Future Directions

This paper has presented the design of a process history capture system as a logging mechanism for distributed processes that are executed over Delta-Enabled Grid Services. We have implemented the PHCS to support rollback of a failed process and evaluation of process failure recovery interferences. This research contributes towards building a robust execution environment for distributed processes executing over autonomous, heterogeneous resources. A unique aspect of this research is the provision of a complete process execution history based on deltas for analyzing data dependencies among concurrently executing processes. The dependency information is not only used for failure recovery of a failed process, but also to evaluate the impact on other processes.

Our future research directions include formalization of the DeltaGrid system using an abstract process execution model, providing a theoretical foundation for building a semantically correct service composition environment. The abstract model will formally define process execution semantics, and present process failure recovery interference rule specification as the basis for user-defined correctness.

References

1. *Specification: Web Services Transaction (WS-Transaction).* (2002) Available from: http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/.
2. *Business Process Modeling Language*. (2002) Available from: http://www.bpmi.org/specifications.esp.
3. *Specification: Business Process Execution Language for Web Services Version 1.1*. (2003) Available from: http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.
4. Ben Abdellatif, T.: *An Architecture for Active Database Systems Supporting Static and Dynamic Analysis of Active Rules Through Evolving Database States*,Ph.D dissertation (1999). Arizona State Univ. *Dept. of Comp. Sci. and Eng.*
5. Blake, L.: *Design and Implementation of Delta-Enabled Grid Services*,M.S. thesis (2005). Arizona State Univ. *Dept. of Comp. Sci. and Eng.*
6. de By, R., Klas, W., Veijalainen, J. , *Transaction Management Support for Cooperative Applications*. (1998): Kluwer Academic Publishers.
7. Eder, J., Liebhart, W.: *The workflow activity model WAMO*, in:*the 3rd international conference on Cooperative Information Systems (CoopIs)*. (1995).
8. Elmagarmid, A., *Database Transaction Models for Advanced Applications*. (1992): Morgan Kaufmann.
9. Foster, I., *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, Int. Journal of Supercomputer Applications, (2001).
10. Garcia-Molina, H., Salem, K. : *Sagas*, in:*ACM International Conference on Management of Data (SIGMOD)*. (1987). pp.249-259.
11. IBM, University of Edinburgh. *OGSA-DAI WSRF 2.1 User Guide*. (2005) Available from: http://www.ogsadai.org.uk/docs/WSRF2.1/doc/index.html.
12. Kamath, M., Ramamritham, K.: *Failure Handling and Coordinated Execution of Concurrent Workflows*, in:*IEEE International Conference on Data Engineering*. (1998). pp.334-341.
13. Liao, N.: *The Extened GridPML Design and Implementation*,M.S. report (2005). Arizona State Univ. *Dept. of Comp. Sci. and Eng.*
14. 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*, in:*the 7th IEEE International Conference on Electronic commerce*. (2005). pp.478-484.
15. 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*, in:*ICEBE*. (2005). pp.433-440.
16. Mikalsen, T., Tai, S., Rouvellou, I.: *Transactional Attitudes: Reliable Composition of Autonomous Web Services*, in:*Workshop on Dependable Middleware-based Systems (WDMS 2002), part of the International Conference on Dependable Systems and Networks (DSN 2002)*. (2002).
17. Oracle. *Oracle9i Streams Release 2 (9.2)*. (2005) Available from: http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96571/toc.htm.
18. Shi, Y., Zhang, L., Shi. B.: *Exception handling of workflow for Web services*, in:*4th International Conference on Computer and Information Technology*. (2004). pp.273-277.
19. Sundermeir, A., Ben Abdellatif, T., Dietrich, S. W., Urban, S. D.: *Object Deltas in an Active Database Development Environment*, in:*the Deductive,Object-Oriented Database Workshop*. (1997). pp.211-229.

20. Tan, Y. *Business Service Grid: Manage Web Services and Grid Services with Service Domain technology*. (2003) Available from: http://www-128.ibm.com/developerworks/ibm/library/gr-servicegrid/.
21. Tartanoglu, F., Issarny, V., Romanovsky, A., Levy, N.: *Dependability in the Web Services Architecture*, in:*Architecting Dependable Systems, LNCS 2677*. (2003).
22. Urban, S.D., Ben Abdellatif T., Dietrich, S. W., Sundermier, A., *Delta Abstractions: A Technique for Managing Database States in Active Rule Processing*, IEEE Trans. on Knowledge and Data Eng., (2003): pp. 597-612.
23. Wachter, H., Reuter, A., *The ConTract Model*, in: *Database Transaction Models for Advanced Applications*, A. Elmagarmid, Editor. (1992). pp. 219-263.
24. Worah, D.: *Error Handling and Recovery for the ORBWork Workflow Enactment Service in METEOR*,M.S. report (1997). University of Georgia. *Computer Science Dept.*
25. Worah, D., Sheth, A., *Transactions in Transactional Workflows*, in: *Advanced Transaction Models and Architectures*, S. Jajodia, and Kershberg,L., Editor, Springer. pp. 3-34.
26. Zeng, L., Lei, H., Jeng, J., Chung, J., Benatallah, B.: *Policy-driven exception-management for composite Web services*, in:*7th IEEE International Conference on E-Commerce Technology*. (2005). pp.355-363.