Using Data Dependencies to Support the Recovery of Concurrent Processes in a Service Composition Environment

Yang Xiao and Susan D. Urban
Arizona State University
School of Computing and Informatics
Department of Computer Science and Engineering
Tempe, AZ 85297-8809
s.urban@asu.edu

Abstract
This paper presents an approach for assessing data dependencies among concurrently executing processes in a service composition environment. The data dependencies are analyzed from data changes that are extracted from database transaction log files and generated as a stream of deltas from Delta-Enabled Grid Services. The deltas are merged by timestamp to create a global schedule of data changes that, together with the process execution context, are used to identify processes that are read and write dependent on failed processes. Process interference rules query the global delta schedule and test application semantic conditions to determine if dependent processes also need to be recovered. This paper focuses on how data dependencies from Delta-Enabled Grid Services are used to construct process dependency graphs for ordering the execution of process interference rules and recovery procedures of dependent processes. We also discuss our results with simulation and evaluation of the concurrent process recovery algorithm.

1. Introduction
Distributed processes that execute over Web Services and Grid Services pose new challenges for the semantic correctness of concurrent process execution. Serializability and compositional serializability [17] cannot be used as correctness criteria since individual service invocations in a process 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.

From an application point of view, dirty reads and dirty writes do not necessarily indicate an incorrect execution, and a relaxed form of correctness dependent on application semantics can produce better throughput and performance. Compensation can be used 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 interference [24, 27]. A robust service execution environment should recover a failed process and effectively handle process interference based on data dependencies and application semantics.

This paper presents an approach to resolving service execution failure, service recovery, and concurrent process interference based on read/write dependency tracking for service execution. Our research has been conducted in the context of the DeltaGrid project [24], which focuses on building a semantically-robust execution environment for processes that execute over Grid Services. The enabling technology for our approach is Delta-Enabled Grid Services (DEGS) [3, 20]. A DEGS is a Grid Service that has been extended with the capability of recording incremental data changes, known as deltas, using features such as Oracle Streams for monitoring database log files. Deltas are generated as a stream of information out of a Grid Service to a Process History Capture System (PHCS) [25] that dynamically merges multiple streams of deltas to create a time-sequenced global schedule of data changes. The global delta object schedule, combined with the process execution context, is used to determine read and write dependencies when a process fails and thus determine when a dependent process may be affected by the recovery of a failed process. A unique aspect of the recovery procedure is that once a dependent process is identified, the dependent process can invoke process interference rules that query the global delta object schedule to test user-defined correctness conditions that determine if the dependent process can continue running or invoke its own compensating procedures.

This paper first summarizes the concept of DEGS, the service composition model, and the process
dependency model [24-27] that we have developed to provide the necessary context for our research. We then present the concurrent process recovery algorithm, demonstrating the use of data dependencies and process interference rules. When a process fails, our approach analyzes the global schedule of data changes to construct a process dependency graph, indicating the other processes that are directly and indirectly dependent on data of the failed process. Recovery of the failed process, typically through compensation, suspends the execution of dependent processes and invokes process interference rules that apply semantic correctness conditions to determine if dependent processes need to be recovered as well. Our work demonstrates the use of process interference rules and the manner in which the process dependency graph is used to order the execution of rules and recovery procedures for dependent processes. We also discuss our results with simulation and evaluation of the concurrent process recovery algorithm. Our research contributes to the development of a semantically robust, concurrent process execution environment for distributed, autonomous service composition and is the first work with service composition to resolve the impact of failure recovery of one process on other concurrently executing processes using delta-based dependency tracking [24].

The remainder of the paper is structured as follows. After presenting related work in Section 2, Section 3 provides an overview of the DeltaGrid environment. Section 4 summarizes the service composition and process dependency models of our research and presents several case studies to illustrate the recovery issues that we are addressing. Section 5 presents the concurrent recovery algorithm together with the use of process interference rules and process dependency graphs. The paper concludes with a discussion of the simulation and evaluation of our work in Section 6, followed by a summary and discussion of future research in Section 7.

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 [6]. Advanced transaction models have been proposed to better support LRTs in a distributed environment [7, 9], including the Nested Transaction Model, the Open Nested Transaction Model, Sagas, the Multi-level Transaction Model and the Flexible Transaction Model. Most of these models do not adequately address recovery for dependent transactions 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 [23]. 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 [21], the Workflow Activity Model [8], the CREW Project [12], the METEOR Project [23], and Units of Work [1]. These projects improve the robustness of transactional workflows, but dealing with dependencies between concurrently executing processes in recovery procedures is still an open issue.

In the Web Services platform, WS-Coordination [5] and WS-Transaction [4] 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) [15] introduces Transactional Attitudes, where service providers and clients declare their individual transaction capabilities and semantics. Web Service Composition Action (WSCA) [18] allows a participant to specify actions to be performed when other Web Services in the WSCA signal an exception. An agent based transaction model [11] integrates agent technologies in coordinating Web Services to form a transaction. Tentative holding is used in [14] to achieve a tentative commit state for transactions over Web Services. Acceptable Termination States [2] 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 supports relaxed isolation and application-dependent semantic correctness for concurrent process execution, with a rule-based approach to resolving the impact of process failure and recovery on other concurrently executing processes. Instead of statically specifying dependency, this research capitalizes on the ability to capture streaming data changes from database log files and to use this information to dynamically analyze write dependency and potential read dependency among concurrently
executing processes, thus providing a more intelligent approach to failure recovery in an autonomous service composition environment.

3. Overview of the DeltaGrid Environment

The results presented in this paper are based on our research with Delta-Enabled Grid Services (DEGS) [3, 20]. A DEGS is a Grid Service that has been enhanced with an interface that provides access to the incremental data changes, or deltas, that are associated with service execution in the context of globally executing processes.

A DEGS uses an OGSA-DAI Grid Data Service [10] 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 [16]. 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 that is local to the service. Deltas are then generated as a stream of XML data from the delta repository and communicated to the delta event processor of the DeltaGrid environment.

Deltas are forwarded from the event processor to a Process History Capture Systems (PHCS) [25]. The PHCS forms a complete execution history for distributed, concurrent processes that are composed of Delta-Enabled Grid Services. The execution history includes deltas from distributed DEGSs and the process runtime context generated by the process execution engine. Deltas are dynamically merged using timestamps as they arrive in the PHCS to create a time-ordered schedule of data changes from distributed DEGSs. This global delta object schedule is used to support recovery activities within the Process Recovery System (PRS) when process execution fails [24]. The global delta object schedule can be used to support the backward recovery of a completed service, which is a process that we refer to as delta-enabled rollback (or DE-rollback) [26]. DE-rollback can be used as long as the traditional notion of recoverability is satisfied (i.e., no other process has read or written data of the process to be rolled back) [13]. The global delta object schedule also provides the basis for discovering data dependencies among processes. The research in [25] describes the storage and indexing components of the PHCS and how they are used to access and analyze data dependencies in support of recovery procedures.


This section provides a more detailed illustration of the data consistency problems that exist with the concurrent execution of processes that are composed of autonomous Grid Services.

4.1. Service Composition and Process Dependency Models

The DeltaGrid service composition model defines a hierarchical service composition structure and the semantics of operation execution failure recovery [26]. The model contains the following execution entities:

- **Operation**: A DEGS service, denoted as $\text{op}_i$
- **Compensation**: An operation used to undo the effect of a committed operation, denoted as $\text{cop}_i$
- **Contingency**: An operation used as an alternative to a failed operation ($\text{top}_i$), denoted as $\text{top}_i$
- **DE-Rollback**: The 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 $\text{deop}_i$
- **Atomic Group**: An execution entity composed of a primary operation ($\text{op}_i$), an optional compensation ($\text{cop}_i$), and an optional contingency operation ($\text{top}_i$), denoted as $\text{ag}_i = \langle \text{op}_i, \text{cop}_i, \text{top}_i \rangle$
- **Composite Group**: An execution entity composed of multiple atomic groups or other composite groups. A composite group can have an optional compensation and an optional contingency, denoted as $\text{cg}_i = \langle \text{ag}_1, \ldots, \text{ag}_n \rangle$
- **Process**: A top level composite group

To support the discussion of concurrent process execution and recovery issues, here we provide an informal overview of concepts in the process dependency model. Formal definitions of the process dependency model appear in [27].

As described in Section 3, the global delta object schedule is used to identify the active processes that are dependent on a failed process. A process-level write dependency exists if a process $\text{p}_i$ writes an object $x$ that has been written by another process $\text{p}_j$ before $\text{p}_j$ completes ($i \neq j$). An operation-level write dependency exists if an operation $\text{op}_i$ of process $\text{p}_i$ writes an object that has been written by another operation $\text{op}_j$ of process $\text{p}_j$. Operation-level write dependency can exist between two operations within the same process ($i = j$). The operations that are write dependent on a specific operation $\text{op}_i$ form $\text{op}_i$’s write dependent set. If $\text{op}_k$ is write dependent on $\text{op}_i$, the enclosing process of $\text{op}_k$ is also write dependent on $\text{op}_i$. 
Similar definitions exist to define read dependencies, but since a DEGS does not capture read information, the global execution context can be used to reveal potential read dependency among operations. An operation \( \sigma k \) is potentially read dependent on another operation \( \sigma j \) under the following conditions:

1) \( \sigma k \) and \( \sigma j \) execute on the same DEGS, and
2) the execution duration of \( \sigma k \) and \( \sigma j \), or \( \sigma k \) is invoked after the termination of \( \sigma j \).

The operations that are potentially read dependent on an operation \( \sigma j \) form a set referred to as \( \sigma j \)’s read dependent set. Potential read dependency can be defined at the process or operation levels.

Note that DE-rollback of an operation is only performed if the operation’s write dependent set is empty.

### 4.2. Case Studies

Consider an online shopping application. The process \( \text{placeClientOrder} \) invokes services that place client orders and thus decrease the inventory quantity. The \( \text{placeVendorOrder} \) process calculates restocking needs based on backorders and quantity on hand and then generates vendor orders. The process \( \text{replenishInventory} \) invokes services that increase the inventory quantity when vendor orders are received. The \( \text{replenishInventory} \) process also processes items on backorder, thus immediately decreasing the inventory for backorders after the inventory has been replenished from the receipt of a vendor order.

Figure 1 presents a scenario where two instances of the process \( \text{placeClientOrder} \) \( (p_{c1} \) and \( p_{c2} \)\) and an instance of the process \( \text{replenishInventory} \) \( (p_i) \) are concurrently executing. The top part of the diagram shows the executed operations of three process instances in time sequence order, and the bottom part shows a graphical view of the global execution history, including deltas generated by an operation’s execution. Each rectangle in a process represents the invocation of a Delta-Enabled Grid Service. To keep the case simple, we use three objects \( (I \) from DEGS1, \( CA \) and \( CB \) from DEGS2) to demonstrate the existence of write dependency. All three process instances access the same inventory item identified by object \( I \). Process instances \( p_{c1} \) and \( p_{c2} \) have created two different orders, identified as \( CA \) and \( CB \).

As shown in Figure 1, \( p_{c1} \), \( p_{c2} \), and \( p_i \) start at different times. \( p_{c1} \)’s current operation is \( \text{packOrder} \) and \( p_{c2} \)’s current operation is \( \text{declinventory} \). \( p_i \) is in the process of backward recovery since the items that have entered the inventory by operation \( \text{inclinventory} \) are recalled by the vendor due to quality problems. \( p_i \)’s recovery procedure contains compensating operations \[ \text{cop:unpackBackOrder, cop:inclinventory, cop:declinventory} \].

The global execution history indicates that:

1) write dependency exists among \( p_{c1} \), \( p_{c2} \), and \( p_i \) for operations that have modified the inventory item \( I \). At the process level, \( p_{c2} \) is write dependent on \( p_{c1} \) and \( p_i \) is write dependent on \( p_{c1} \) and \( p_{c2} \), and \( p_{c1} \) is not write dependent on \( p_{c2} \) or \( p_i \).

2) due to the existence of write dependent processes on \( p_{c1}:\text{inclinventory} \) and \( p_{c2}:\text{declinventory} \), DE-rollback cannot be performed on \( p_{c1}:\text{inclinventory} \) and \( p_{c2}:\text{declinventory} \) if these two operations do not have predefined compensation.

3) \( p_{c2}:\text{declinventory} \) modifies the inventory item \( I \) that has been written by \( p_{c1}:\text{inclinventory} \) and \( p_{c2}:\text{declinventory} \). The process \( p_{c2} \) is therefore operating under the assumption that there are enough items in inventory to process the order. After \( p_i \) executes its compensating procedures, there may no longer be enough items in inventory to process the order for \( p_{c2} \). If the recovery procedure of \( p_i \) does affect \( p_{c2} \)’s execution, then process interference exists between \( p_i \) and \( p_{c2} \) and \( p_{c2} \) needs to be recovered as well. Otherwise \( p_{c2} \) can keep running, ignoring the changes made by the recovery procedure of \( p_i \).

Now consider a scenario involving potential read dependency. Figure 2 presents a scenario where two instances of the process \( \text{placeClientOrder} \) \( (p_{c1} \) and \( p_{c2} \)\) and an instance of the process \( \text{placeVendorOrder} \) \( (p_r) \) are concurrently executing. Same as the write dependency scenario, these process instances are all related to inventory item \( I \).

After \( p_{c1} \) and \( p_{c2} \) withdraw inventory items from the warehouse, \( p_r \) checks the available stock of inventory item \( I \) through operation \( \text{calcRestockNeed} \), and proceeds with the next operation \( \text{calcBackorder} \). At this time, the client of order \( CA \) cancels the order, which causes \( p_{c1} \) to be backward recovered. The compensation activities \[ \text{cop:unpackOrder, cop:inclinventory, cop:declinventory} \] are executed.

Operations \( p_{c1}:\text{declinventory} \), \( p_{c2}:\text{declinventory} \) and \( p_r:\text{calcRestockNeed} \) have overlapping execution on the same DEGS site \( \text{DEGS1} \) and therefore \( p_r:\text{calcRestockNeed} \) is potentially read dependent on \( p_{c1}:\text{declinventory} \) and \( p_{c2}:\text{declinventory} \). \( p_r \)’s execution could potentially be affected by \( p_{c1} \)’s recovery procedure since the compensation of \( p_{c1} \) has modified the inventory quantity which is used by \( p_r \) to calculate the restock need. As a result, \( p_r \) could potentially place an order that could lead to significant overstocking.

In the write dependency scenario, if the recovery of \( p_{c1} \) does affect \( p_r \)’s execution (i.e., \( p_{c1} \) was a large order), \( p_r \) needs to be recovered. Otherwise \( p_r \) can keep running.
Whether the recovery of \( p_e \) does affect \( p_r \)’s execution is dependent on application semantics.

### 5. Concurrent Recovery Algorithm

Given that we can collect and analyze data dependencies from the DEGS that compose a process, this section outlines the concurrent recovery algorithm. We first define the concept of process interference rules (PIRs). We then provide a sketch of the concurrent recovery algorithm and illustrate some of the execution issues that must be considered.

#### 5.1. Process Interference Rules

PIRs are invoked for processes that are read and/or write dependent on a failed process. PIRs test user-defined conditions that determine if the dependent process should continue running or invoke its own recovery procedures. A PIR is written from the perspective of an executing process (\( p_e \)) that is interrupted by the recovery of an unknown failed process (\( p_f \)). The interruption occurs because \( p_e \) is identified from the global delta object schedule as being write dependent or potentially read dependent on \( p_f \). PIRs are expressed using an extended version of the Integration Rule Language (IRL) [19], which was originally defined to provide a rule-based approach to component integration.

A process interference rule has four elements: event, define, condition, and action. The triggering event of a PIR is a write or a read dependent event. A write dependent event is triggered after the backward recovery of a failed process for every write dependent process, in the format of:

\[
\text{<writeDependentProcessName>} \text{WriteDependency}(\text{failedProcess}, \text{wdProcess})
\]

The write dependent event contains the name of the write dependent process instance (\( \text{writeDependentProcessName} \)), and two parameters: the identifier of the failed process (\( \text{failedProcess} \)) and the identifier of the write dependent process instance (\( \text{wdProcess} \)). As an example, in Figure 1, after the compensation of the failed process replenishInventory (\( p_r \)), a write dependency event placeClientOrderWriteDependency (\( p_e, p_{c2} \)) will be raised. Similarly, a read dependent event is
raised for every potentially read dependent process, in the format of `<readDependentProcessName>`

\[ \text{ReadDependency}(\text{failedProcess}, \text{rdProcess}) \]

Define declares variables to support condition evaluation. A condition is a Boolean expression to determine the existence of process interference based on application semantics. Action is a list of recovery commands. The command could invoke backward recovery of a process (i.e., Compensation), re-execution of a process, backward recovery of an operation, or re-execution of an operation.

The global execution history provides an object model as an interface to access delta values and other execution information in declare and condition clauses of a PIR. For example, the method `getDeltas(className)` is used to retrieve all the deltas of a given class name that are created by an operation or a process. `getDeltas(className, attrName)` further limits the returned deltas to be of a given attribute name identified by `attrName`. As additional examples, `getDeltasBeforeRecovery` returns deltas that are created by a process before any recovery activity is performed, and `getDeltasByRecovery` returns deltas that are created by the recovery activities of a process.

In Figure 1, recall that process `placeClientOrder (p_c2)` is write dependent on `replenishInventory (p_i)`. As a result, `p_c2` may process an order for a client based on an increase in inventory caused by `p_i`. If `p_i` fails, however, after increasing the inventory for an item `x` that is also accessed in `p_c2`, then `p_c2` may also be affected if there will no longer be enough items in inventory for item `x`. As a result, `p_c2` may need to be backward recovered.

Figure 3 presents a PIR expressing the above application constraint. In this scenario, compensation of the process `replenishInventory (p_i)` raises the event `placeClientOrderWriteDependency (p_c2, p_i)` which matches the event defined in the rule. The define clause creates bindings for inventory items (`decreasedItems`) that have been decreased by the recovery of `failedProcess`. The select statement finds deltas related to the quantity of an inventory item (`getDeltasByRecovery ("InventoryItem", "quantity")`) created by the recovery activities of `failedProcess`, finding items with decreased quantity by accumulating the changes on quantity for each item.

The condition has a when statement checking to determine if any decreased item (`declitem`) appears in the current client order. The evaluation is conducted by checking if any `declitem` is the same as the old of deltas associated with class `InventoryItem` and attribute `quantity` created by `wdProcess`. The action is to backward recover `wdProcess` from the current operation, if the when statement evaluates to true.

Similar rules can be developed to check user-defined conditions in the case of potential read dependencies. In this case, a PIR can be used to 1) determine if the failed process intersects with the potentially read dependent process on critical data items by querying deltas on the process execution history, and 2) express application-specific conditions on the status of the critical data items after recovery of the failed process. If application constraints are violated, the read dependent process can invoke recovery procedures. Otherwise the read dependent process can resume execution. More examples of PIRs for read and write dependencies can be found in [24].

Create rule

\[ \text{InventoryDecrease} \]

Event `placeClientOrderWriteDependency (\text{failedProcess}, \text{wdProcess})`

Define `decreasedItems` as

\[ \text{select item} \text{Id} \text{old} \]
\[ \text{from \text{failedProcess}getDeltasByRecovery ("InventoryItem", "quantity")} \]
\[ \text{group by \text{item} \text{Id}} \]
\[ \text{having sum (\text{new value} - \text{old value})} < 0 \]

Condition when `exists declitem in decreasedItems`

\[ \text{declitem} \text{Id} \]
\[ \text{select} \text{dId} \]
\[ \text{from \text{declitemProcess}getDeltasByRecovery ("InventoryItem", "quantity")} \]

Action `deepCompensate (\text{wdProcess})`

Figure 3. PIR `decreaseInventory for Process placeClientOrder`

5.2. Recovery Algorithm

When a process `p_i` fails, PIRs are evaluated on dependent processes, including write dependent processes and potentially read dependent processes. If a PIR evaluates to true, a dependent process `p_d` is recovered due to process interference. The backward recovery of `p_d` might cause a cascaded recovery on one of `p_d`’s dependent processes `p_dd`, although `p_dd` is not directly dependent on `p_i`. In this case, `p_dd` is indirectly dependent on `p_i` through `p_d`. Thus cascaded recovery must be considered in a concurrent process execution environment.

To handle cascaded recovery, a Process Dependency Graph (PDG) is constructed for the failed process `p_i` containing processes with a PIR that are directly and indirectly dependent on `p_i`. A PDG for `p_i` is a directed acyclic graph with `p_i` as the root of the graph. A PDG provides a snapshot of the dependency relationships associated with the failed process.

5.2.1 PDG Construction

In a PDG, each node represents a process. The outgoing edge of a node `p_i` points to another node `p_j` if `p_j` is dependent on `p_i`. Figure 4 presents the structure of a node in a PDG. A `PDGNode` represents a process `p_i` with `pId` as identifier, and dependency relationships with other nodes. A `PDGNode` has a `backwardRecCmdList` to store backward recovery commands, and a
forwardRecCmdList to store p_i’s forward execution commands. The attribute children is a list that contains all of the processes that are dependent on p_i, and parent is a list containing all of the processes on which p_i depends. The operation isAncestorOf(PDGNode, node) to used to determine if a node is a descendant of PDGNode.

<table>
<thead>
<tr>
<th>PDGNode</th>
</tr>
</thead>
<tbody>
<tr>
<td>pld : string</td>
</tr>
<tr>
<td>backwardRecCmdList : object</td>
</tr>
<tr>
<td>forwardRecCmdList : object</td>
</tr>
<tr>
<td>children : object</td>
</tr>
<tr>
<td>parent : object</td>
</tr>
<tr>
<td>pirEvalResult : Boolean</td>
</tr>
<tr>
<td>isAncestorOf() : Boolean</td>
</tr>
</tbody>
</table>

**Figure 4. PDG Node Structure**

Since backward recovery of a process p_i raises read and write dependency events on each of p_i’s dependent processes p_j, if p_i has a matching PIR, a PIR instance will be evaluated based on p_i. It is possible that p_j has read or write dependency on multiple processes (p_j has multiple parents), thus multiple instances of a PIR will be evaluated for p_i based on different parents. p_i can also potentially have a PIR for read dependency and a PIR for write dependency. In this case, p_i will be recovered if any of its PIR instances evaluates to true, but the action must be deferred until all of the associated PIR instances are evaluated. To enforce this policy, a PDGNode has an attribute pirEvalResult of type Boolean to store the evaluation result of all of the PIR instances associated with p_i. If p_i has one parent, pirEvalResult is the evaluation result of the PIR instance based on that parent. If p_i has multiple parents, pirEvalResult is the accumulated evaluation result which is the disjunction of each PIR instance evaluation. If any of p_i’s PIR instances evaluates to true, p_i’s pirEvalResult is true.

In a concurrent process execution environment with relaxed isolation, two processes might be dependent on each other. In this case, each process waits for the PIR evaluation result from the other, causing a potential deadlock situation. Thus during PDG construction, if two processes are dependent on each other, a PIR evaluation order is enforced on these two processes based on the distance of the dependency relationship between each process and the failed process. Applying the PIR evaluation order, cyclic dependency between two processes is removed in a PDG, thus ensuring termination in the traversal of the PDG.

Figure 5 demonstrates the procedure of enforcing the PIR evaluation order on two processes that are dependent on each other. Assume p_i is the failed process. Figure 5(a) shows the actual dependency information retrieved from the global execution history, which indicates that processes p_2 and p_4 are dependent on each other. During PDG construction, after processes dependent on p_i (p_2 and p_4) are retrieved, p_2 is not added as a dependent process of p_i in PDG since p_2 is already in PDG as a node with a closer dependency on p_i, as shown in Figure 5(b). Thus the cyclic dependency relationship between p_2 and p_4 is removed during PDG construction, and p_2’s PIR evaluation will be conducted before p_i’s PIR evaluation.

**Figure 5. Removal of Cyclic Dependency During PDG Construction**

An important property of a PDG is that it guarantees that updates are not lost in the recovery process, since 1) compensation (i.e. logical rollback) has higher priority than DE-rollback, and 2) DE-rollback (i.e. physical rollback) is only performed in the case when a process has no write dependent processes. Although a PDG provides a snapshot of process dependency relationships at the moment of failure occurrence, during PDG node processing, the semantic conditions for DE-rollback are dynamically evaluated, considering write dependencies that are also created by recovery procedures. For example, in Figure 5, although p_2 is removed as a dependent process of p_i in the PDG, the recovery procedure of p_i retrieves the complete read/write dependency information from the global execution history, including information about p_2. If p_2 is write dependent on p_i, p_i can only perform compensation (and not DE-rollback) as a logical recovery procedure.

**5.2.2 Multi-Process Recovery**

This section presents an example to illustrate the cascaded recovery procedure of the multi-process recovery algorithm. The complete pseudo code for the concurrent recovery algorithm can be found in [24].

The process execution engine retains an executeQ to hold operations from processes in normal execution. When a process p_i fails, there are two types of processes in executeQ: processes dependent on p_i (in
p’s PDG), and processes that are not dependent on p. Dependent processes are suspended for PIR evaluation, while independent processes remain in the executeQ and continue normal execution.

Figure 6 shows a sample PDG (PDGi) containing nine concurrently executing processes identified as p₁–p₉. Each arrow points to the parent of a node, indicating the process that a node depends on. A check mark indicates that the condition of a PIR instance for a node evaluates to true, meaning the node needs to be recovered according to the action of the PIR. A cross mark indicates that the condition of a PIR instance for a node evaluates to false, meaning the node can keep running. PDGₙ demonstrates the handling of nodes with multiple parents (p₆ and p₉), processing nodes with a PIR that evaluates to true (p₅, p₆, p₇, p₈, p₉, and p₁), and processing nodes with a PIR that evaluates to false (p₃ and p₈). Table 1 illustrates the execution of the multi-process recovery algorithm for PDG₁. In Table 1, level indicates the current processing level of PDG₁. bkRecList stores all of the nodes to be backward recovered at this level. pi indicates the current node (in bkRecList) whose backward recovery is invoked. pj indicates the current child node of pi whose PIR is being evaluated. PIR evaluation shows the evaluation result of PIR on pi. pi.parent shows the current parent of pi since a node’s parent might be changed after PIR is evaluated on its parents. tmpBKRecList stores the next level of nodes to be backward recovered. executeQ shows the nodes that are added back to executeQ.

The procedure starts from level 1, adding the failed process pi into bkRecList, and evaluating the PIR on each child of pi. p₂ and p₄ are children with a single parent, with a PIR that evaluates to true, thus they are added to tmpBKRecList as nodes to be recovered at level 2. Since PI values to false on p₁, p₁ is returned to executeQ, and the descendents of p₁ are processed by invoking processChildrenOfProcessWithFalsePIR(p₁). This function releases p₁ to executeQ, and removes p₁ from p₁.parent and p₁.parent. At this time, we cannot determine whether p₈ and p₉ need recovery since they are dependent on other parents, p₂ and p₄, respectively. After processing completes on level 1, the nodes in tmpBKRecList are added to bkRecList to process level 2 of PDG₁. After executing backward recovery on p₂, then p₁ and p₄’s PIRs evaluate to true. Since p₈ and p₉ have a single parent pi, p₈ and p₉ are added to tmpBKRecList. Notice that p₈ becomes a node with a single parent after function processChildrenOfProcessWithFalsePIR(p₃) removes p₃ as p₈’s parent. Similarly, after backward recovery of p₄, p₈ is added to tmpBKRecList, and p₈ is added to executeQ. In level 3, backward recovery is invoked for p₅, p₆ and p₇. The procedure ends since the entire PDG has been processed.

Table 1. Cascaded Recovery Process for PDG₁

<table>
<thead>
<tr>
<th>Level</th>
<th>bkRecList</th>
<th>pi</th>
<th>pi ∈ pi.child</th>
<th>PIR eval</th>
<th>pi.parent</th>
<th>tmpBKRecList</th>
<th>executeQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[p₁]</td>
<td>p₁</td>
<td>p₁ true</td>
<td>[p₁]</td>
<td>[p₁]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>p₁ false</td>
<td>[p₁]</td>
<td>[p₁]</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>n/a</td>
<td>[p₁]</td>
<td>[p₁]</td>
<td>p₁</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>n/a</td>
<td>[p₁]</td>
<td>[p₁]</td>
<td>p₁</td>
<td>p₁</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>true</td>
<td>[p₁]</td>
<td>[p₁, p₂]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>[p₁, p₂]</td>
<td>p₁</td>
<td>p₁ true</td>
<td>[p₁]</td>
<td>[p₂]</td>
<td>[p₁]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>p₁ true</td>
<td>[p₁]</td>
<td>[p₂]</td>
<td>[p₁, p₄]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>p₁ true</td>
<td>[p₁]</td>
<td>[p₂, p₄]</td>
<td>[p₁]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>p₁ false</td>
<td>[p₁]</td>
<td>[p₂]</td>
<td>p₁</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>p₁ true</td>
<td>[p₁]</td>
<td>[p₂]</td>
<td>[p₁]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>p₁ true</td>
<td>[p₁]</td>
<td>[p₂]</td>
<td>[p₁, p₄]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>p₁ false</td>
<td>[p₁]</td>
<td>[p₂]</td>
<td>p₁</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>p₁ true</td>
<td>[p₁]</td>
<td>[p₂]</td>
<td>[p₁, p₄]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>p₁ false</td>
<td>[p₁]</td>
<td>[p₂]</td>
<td>p₁</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>p₁ true</td>
<td>[p₁]</td>
<td>[p₂]</td>
<td>[p₁, p₄]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p₁</td>
<td>p₁ false</td>
<td>[p₁]</td>
<td>[p₂]</td>
<td>p₁</td>
<td></td>
</tr>
</tbody>
</table>

This example demonstrates how a failed process and its dependent processes are recovered in a concurrent process execution environment. If multiple processes fail, a PDG is constructed for each failed process, in the order of failure occurrence. For example, assume a process p₃ fails before a process p₁ fail. A PDG for p₁ (PDGₙ) is constructed before the PDG for p₃ (PDGₙ). Assume the two processes have a common dependent process p₁. If p₈ is recovered during PDGₙ’s processing, p₈ will not appear in PDGₙ, but any write dependencies introduced by the execution and recovery of p₈ will be considered in the evaluation of DE-rollback applicability in PDGₙ (since all data changes are captured in the global delta object schedule that is used to determine the write dependent set of p₈). If p₈ is not affected by p₁’s recovery and is put back into executeQ, p₁ is an active process in normal execution, and p₈ will appear in PDGₙ.
6. Performance Evaluation

To evaluate the performance of the PRS, we developed a simulation framework that combines the PHCS and the PRS that were implemented as part of this research with other simulated components of the DeltaGrid environment. The simulation framework was developed using the DEVSJAVA framework [28], a modeling and simulation tool for discrete event system specification. In the remainder of this section, we summarize the results of the performance evaluation for the PRS recovery command generation and multi-process recovery.

The recovery command generation time is affected by 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. Recovery command generation time is also affected by 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 (medium) and 100–1000 (large). To vary a process’s complexity, we tested on processes with nesting levels from 1-5. 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 occurs because of the need to increase write dependency retrieval time for nested processes. The recovery command generation time has a linear increase when the number of concurrent processes grows.

We also evaluated the PDG construction time and cascaded recovery processing time. PDG construction time is important since the execution engine suspends normal processing during PDG construction. PDG construction is affected by the number of concurrent processes and process dependency density, which is the ratio between the number of processes in the PDG and the total number of concurrent processes. The evaluation was conducted by varying concurrency and process dependency density. To vary the number of concurrent processes, we tested on two different ranges: 10–100 (medium) and 100–1000 (large). To vary a process’s dependency density, we tested on different process dependency density ranges: < 10%, 30%~50%, 50%~70%, and > 90%.

Table 2 illustrates the impact of process dependency density on PDG construction time.

### Table 2. PDG Construction Time under Different Dependency Density

<table>
<thead>
<tr>
<th>Number of concurrent processes</th>
<th>PDG construction time (Millisecond)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>200000</td>
</tr>
<tr>
<td>100</td>
<td>220000</td>
</tr>
<tr>
<td>1000</td>
<td>240000</td>
</tr>
<tr>
<td>10000</td>
<td>260000</td>
</tr>
</tbody>
</table>

7. Summary and Future Work

This paper has presented a novel approach for assessing data dependencies among concurrently executing processes in a service composition environment, using delta-based dependency tracking to resolve the impact of the failure recovery of one process on other concurrently executing processes. With the prevalence of streaming applications as well as new database capabilities that allow applications to externalize the contents of database log files, our approach provides a more intelligent way of automatically detecting data dependencies to determine the affect of process failure and recovery procedures. Our results demonstrate a more optimistic approach to service execution that relies on application semantics to assess the correctness of concurrent process execution.

One direction for future research involves further investigation of the specification of process interference rules. The use of PIRs needs to be addressed in the larger methodological context of specifying dynamic processes for the composition of Web and Grid Services. Another direction involves the
investigation of a distributed PHCS to support process recovery in a dynamic service composition environment. Our current work focused on the theoretical aspects of using deltas from distributed sites to analyze data dependencies. The next generation of the DeltaGrid system will be based on a peer-to-peer mode where each DEGS has as an active agent that manages deltas locally and communicates with other agents to resolve global data dependencies when process failure occurs.

8. References