Recovery of Concurrent Processes using Rules and Data Dependencies

Ashwini B. Shinde  
Department of Information Technology  
Gharda Institute Of Technology, Lavel, Chiplun, Maharashtra, India

Akshaya A. Bhosale  
Department of Information Technology  
Gharda Institute Of Technology, Lavel, Chiplun, Maharashtra, India

Abstract- This paper presents a recovery algorithm for service execution failure in the context of concurrent process execution. The recovery algorithm was specifically designed to support a rule-based approach to user-defined correctness in execution environments that support a relaxed form of isolation for service execution. 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 time stamp 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 are used to express semantic conditions that determine if a process that is dependent on a failed process should recover or continue execution. The recovery algorithm integrates a service composition model that supports nested processes, compensation, contingency, and rollback procedures with the data dependency analysis process and rule execution procedure to provide a new approach for addressing consistency among concurrent processes that access shared data. We present the recovery algorithm.

Keywords – Service composition, relaxed isolation, concurrent process recovery, transaction processing.

I. INTRODUCTION

Web services and service-oriented computing has changed data access patterns for distributed computing environment, which develop processes that are composed of distributed service executions. Some challenges for the semantic correctness of concurrent process execution have been faced by processes which executes over web services and Grid services. For individual services, they have their own local transaction semantics, where commit operation of service is controlled by the residing service. It results in commit of service before completion of process. Due to relaxation of isolation in service execution of a process results in dirty read and write[1].

To recover from dirty read and write compensation is undo to undo a process. Compensation procedure affects some concurrently executing processes, to identify this dependency Process Interference Rules (PIR)[8] are used. This ability of capturing and analyzing data dependencies is absent in current service-oriented architectures.

To resolve problem related to execution failure, service recovery, and concurrent process interference based on read/write dependency DeltaGrid projects are used, which create a semantically robust execution environment for processes that execute over Grid Services. For this we are using Delta-Enabled Grid Services (DEGS) [5][9][10]. A DEGS records data changes in terms of deltas. Deltas are used as input to a Process History Capture System (PHCS), which merges multiple streams of deltas to create time-sequenced global schedule of data changes. This global schedule is used to construct a process dependency graph (PDG) [7], which gives idea about the processes that are directly and indirectly dependent on data of the failed process. After indentifying dependencies PIR[8] apply semantic correctness conditions to determine whether recovery is required for dependent process or not. Some concepts like rollback, compensation, forward recovery and logging have been used to achieve recovery from failure.

II. RELATED WORK

A. Delta-Enabled Grid Services

DEGS provides access to the incremental data changes associated with services execution in the context of globally execution process. A DEGS uses an OGSA-Dai Grid Data Services for data interaction, modifying SQLUpdate activity feature for database access to provide the functionality necessary for capturing and returning delta values. A DEGS uses the object delta concept to create a conceptual view of relational deltas. A tuple of relation can be associated with an instance of DeltaObject. A DeltaObject has a className (relation) to which object belongs, and
objectId (primary key) to uniquely identify the associated object instance. A DeltaObject can have multiple DeltaProperty object. Each DeltaProperty object has a PropertyName, and one or more PropertyValue(delta values). A PropertyValue contains an oldValue and a newValue. Each PropertyValue is associated with a DataChange object; it has a processId and an operationId, indicating the global process and operation that has created the PropertyValue, with timestamp to record the actual time of change. The object structure shown in figure 1.

![Diagram](attachment:image.png)

**Figure 1: The Delta Structure of a DEGS**

### B. Delta-Enabled RollBack (DE-rollback)

DE-rollback[10] is performed to reverse the data changes that have been introduced by a service execution to their before-execution images, it also used to reverse the result of a service execution even after the execution has terminated. Figure 2 shows the execution of two processes, p1 and p2. Here p1 consist of two services op11 and op12. Process p2 also consist of two services op21 and op22. Both processes access X and Y, with schedule shown as x1, followed by x2 and x3, as well as y1 followed by y2. As shown in figure 2, if DE-rollback is invoked on op22, the object delta created by op22 (y2) will be removed, and the value of Y will be resorted to the value y1. Similarly, DE-rollback can be applied to op21 to reverse the value of X from x3 to x2.

![Diagram](attachment:image.png)

**Figure 2: Delta-Enabled Rollback**

### C. Process Dependency Graph (PDG)

PDG consist of nodes with represents process. The outgoing edge of a node px points to py if px is dependent on py. For each PDG PDGNode is created, which represents a process pi with p Id as identifier, and dependency relationship with other nodes. A PDGNode has a backwardRecCmdList to store backward recovery commands, and a forwardRecCmdList to store pi’s forward execution commands. A children attribute contain list of all the processes that are dependent on pi, and parent attribute contain list of all the processes on which pi depends. The operation isAncestorof(PDGNode,node) is used to determine if a node is a descendant of PDGNode. pirEvalResult of type Boolean to store the evaluation result of all of the PIR instances associated with pi. The structure of PDGNode is shown in figure 3.
In PDG deltas are not lost during the recovery process. The algorithm is generated for PDG which contain all processes that are directly or indirectly dependent on the failed process. The input to the algorithm is failed process \( p_i \).

**D. Process Interference Rule (PIR)**

To recover failed process cascaded recovery of other dependent process is required. In the DeltaGrid environment, PIR is used to determine if the cascaded recovery of dependent process is required or not. PIR is used to specify how to handle concurrent processes that are either read or write dependent on failed process. A PIR has four elements: event, define, condition and action. There are two events in PIR,

1) A write dependent event. The format is,

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

The dependent event contain the name of the write dependent process instance (writeDependentProcessName), and two parameters: the identifier of failed process (failedProcess) and the identifier of write dependent process (wdProcess).

2) A read dependent event. The format is,

\[
<\text{readDependentProcessName}>\text{ReadDependency}(\text{failedProcess}, \text{rdProcess}).
\]
The dependent event contain the name of the read dependent process instance (readDependentProcessName), and two parameters: the identifier of failed process (failedProcess) and the identifier of read dependent process (rdProcess).

The structure of PIR is as shown in figure 5.

Create rule ruleName
  event failureRecoveryEvent
  define [viewName as <OQL expression>
  condition [when condition]
  action recovery commands

Figure 5:-PIR Structure

E. DeltaGrid Abstract Execution Model

Figure 3.2 shows the DeltaGrid abstract execution model[8], is composed of three components: the service composition and recovery model, the process dependency model, and process interference rule. The service composition model defines a hierarchical service composition structure as well as the semantics of execution entities for the handling of operation execution failure occurring at any compositional level. The process dependency model further defines the relationships that exist among concurrently executing processes in terms of write dependencies and potential read dependencies.

Figure 6:DeltaGrid Abstract Execution Model

III. PROPOSED ALGORITHMS

A. Composite Group Deep Compensation Algorithm

Algorithm given in figure 4.1 invokes the deep compensation of an immediately enclosing composite group of subgroups if the contingency of the subgroup fails. The input to the algorithm is the composite group to be deep compensated.

    public void deepCompensate(Compositegroup cg)
    {

Figure 6:DeltaGrid Abstract Execution Model

International Journal of Latest Trends in Engineering and Technology (IJLTET)
//get a list of executed critical subgroups of cgi in reverse execution order
- \( C = \{sg_{ik} | \text{sg}_{ik} \in C \text{ of } c_{gi} \} (k=n..1) \)

//iterate through every executed subgroup of cgi
FOR EACH \( \text{sg}_{ik} \in C \)
{
//check if \( \text{sg}_{ik} \) is an atomic group
CASE:
1. \( \text{sg}_{ik} \) is an atomic group
   EXECUTE post-commitRecoverAtomicGroup(\( \text{sg}_{ik} \));
   CONTINUE;
2. \( \text{sg}_{ik} \) is a composite group:
   //check if \( \text{sg}_{ik} \) has shallow compensation \( c_{gi} \)
   CASE:
   2.1 \( \text{sg}_{ik} \) has \( c_{gi} \):
      EXECUTE \( c_{gi} \);
      //check \( c_{si} \) execution result
      CASE:
      2.1.1 \( c_{sg_{ik}} \) SUCCEEDS:
         CONTINUE;
      2.1.2 \( c_{sg_{ik}} \) FAILS:
         EXECUTE deepCompensation(\( \text{sg}_{ik} \));
      END CASE:
   2.2 \( \text{sg}_{ik} \) has no \( c_{sg_{ik}} \):
      EXECUTE deepCompensation(\( \text{sg}_{ik} \));
      CONTINUE;
   END CASE:
 END CASE:
}

B. Postcommit Recovery of an Atomic Group

The algorithm shown in figure 4.2 checks the recovery attributes of an operation and determine recovery procedure.

public Operation post-commitRecoverAtomicGroup(AtomicGroup \( \text{ag}_{ij} \))
{
    get \( \text{ag}_{ij} \)'s primary operation=>op_{ij};
    //check \( \text{op}_{ij} \)'s post-commit recoverability
    CASE:
    1. \( \text{op}_{ij} \) is COMPENSATABLE:
       //check if \( \text{ag}_{ij} \) compensation \( c_{op_{ij}} \)
       CASE:
       1.1 \( \text{ag}_{ij} \) has \( c_{op_{ij}} \):
          RETURN \( c_{op_{ij}} \);
       1.2 \( \text{ag}_{ij} \) has \( c_{op_{ij}} \):
          //the applicability of \( c_{op_{ij}} \) will be checked before execution
          RETURN \( c_{op_{ij}} \);
       END CASE:
    2. \( \text{op}_{ij} \) is REVERSABLE:
       RETURN \( c_{op_{ij}} \);
    3. \( \text{op}_{ij} \) is DISMISSABLE:
       RETURN NULL:
    END CASE:
}
C. PDG Construction Algorithm

PDG Construction Algorithm shown in figure 3 is consisting of steps required to construct PDG.
1. Create a hashtable (nodeTable) to store the PDG.
2. Create three lists to assist in the construction process:
   2.1. root to iterate through different levels of a PDG,
   2.2. tmpRoot to temporarily store the children of processes in root, and
   2.3. dpList to temporarily store the dependent processes of a single process.
3. Add the failed process pi in root to start PDG construction.
4. For each process pi in root,
   4.1. Add every dependent process pd of pi into dpList.
   4.2. For each pd of dpList
      4.2.1. If pd already has a node in nodeTable, retrieve it. Otherwise, create a node for pd.
      4.2.2. If pd is not an ancestor of pi (no circular dependency), make pd dependent on pi and
         add pd into tmpRoot (creating the process list for the next level of the PDG).
5. After processing all elements of root, empty dpList and move processes from tmpRoot to root. If root is not empty, repeat step 4.

D. Multiprocess Recovery Algorithm

The algorithm shown in figure 4.4 coordinates backward recovery and forward recovery execution for the failed process and dependent processes. The input to the algorithm is failed process. The goal of this algorithm is to identify dependent processes and to evaluate which processes are affected using PIRs to derive proper recovery commands for each process.
1. For each child (pc) of px,
2. For each PIR instance of pc,
   2.1 Evaluate the PIR condition
   2.2 If the PIR condition is true, further verify if px is the only parent of pc. If yes, then pc is added to
      the tmpBKRecList. Otherwise, pc’s pirEvalResult is set to true and px is removed from pc’s parent
      since px’s impact on pc has been processed.
      Then, the recovery commands specified in the PIR action are merged to pc’s
      backwardRecCmdList and forwardRecCmdList.
   2.3 If the PIR condition is false, verify if px is the only parent of pc. If px is the only parent of pc, pc
      is put back into the executeQ and the function processChildrenOfProcessWithFalsePIR is invoked.
      This function traverses the subgraph with pc as the root, processing descendants of pc. The
      function adds any descendant pd of pc back to executeQ if pd has only one parent, and removes pc
      from pd’s parent if pd has multiple parents.

IV. CONCLUSION

It provides more intelligent way of detecting data dependencies to determine the affect of process failure and recovery procedure. This research help to identify data dependencies between concurrent process in a service composition environment with the help of delta-based dependency tracking which resolve the impact of the failure recovery of one process on other concurrently executing processes.

V. FUTURE WORK

This research gives the recovery for concurrent processing groups we can enhance this research for parallel groups and also it is possible to detailed analysis of looping, choice, and parallel control structure.

REFERENCES


Vol. 3 Issue 1 September 2013 6 ISSN: 2278-621X


