A New Language to Support Flexible Failure Recovery for Workflow Management Systems

Gwan-Hwan Hwang[1], Yung-Chuan Lee[1], and Bor-Yih Wu[2]

[1] Dept. of Information and Computer Education, National Taiwan Normal University, Taiwan
{ghhwang,u86278}@ice.ntnu.edu.tw
[2] Software Project Development Division, Stark Technology Inc., Taiwan

Abstract. In this paper, we propose a new failure-recovery model for workflow management systems (WfMSs). This model is supported with a new language, called the workflow failure-handling (WfFH) language, which allows the workflow designer to write programs so that s/he can use data-flow analysis technology to guide the failure recovery in workflow execution. With the WfFH language, the computation of the end compensation point and the compensation set for failure recovery can proceed during the workflow process run-time according to the execution results and status of workflow activities. Also, the failure-recovery definitions programmed with the WfFH language can be independent, thereby dramatically reducing the maintenance overhead of workflow processes. A prototype is built in a Java-based object-oriented workflow management system, called JOO-WfMS. We also report our experiences in constructing this prototype.

1 Introduction

Workflow management systems (WfMSs) are software systems for supporting coordination and cooperation among members of an organization whilst they perform complex business tasks [1–5]. Business tasks are modeled as *workflow processes*, which are then automated by the WfMS. The workflow model (also referred to as *workflow process definition*) is the computerized representation of the business process. It defines the starting and stopping conditions of the process, the activities in the process, and control and data flows among these activities. An *activity* is a logic step within a workflow, which includes the information about the starting and stopping conditions, the users who can participate, the tools and/or data needed to complete this activity, and the constraints on how the activity should be completed. Activities are usually organized into a directed graph that defines the order of execution among the activities in the process. Nodes and edges in the graph represent activities and control flow, respectively. A *workflow process instance* is the execution of a workflow process definition by the WfMS. The execution of a workflow process instance is controlled by the *workflow engine*.

Two different types of problems or anomalous situations can occur during workflow execution: *exceptions* and *failures* [6]. Exceptions are semantic failures that can be caused by a system failure or by a new situation introduced by the external

J. Favela and D. Decouchant (Eds.): CRIWG 2003, LNCS 2806, pp. 135–150, 2003.

environment. A comprehensive approach to the management of exceptions is to integrate the exception handler with the WfMS. The exception-handling mechanism must be able to capture exceptional events and react to them. The WfMS may terminate the workflow process or return to the execution of the workflow process after the exception has been handled [7,8].

The other problem in managing workflow processes is that of *failure recovery* [9–11]. The goal of failure recovery is to bring the failed workflow process back to some semantically acceptable state so that the cause of the failure can be identified. The problem can then be fixed and the execution resumed with the hope that the workflow process will be completed successfully. The basic failure-recovery process includes the following three steps:

1. The execution of the workflow process is terminated and the workflow engine then decides the *end compensation point* (ECP) and the *compensation set* of the occurred failure.
2. Activities in the compensation set are compensated.
3. The workflow process is restarted from the ECP.

An ECP is a previously executed activity of the workflow process which represents an acceptable intermediate execution state where certain actions can be taken so that either the problems which caused the failure can be fixed or the execution path of the workflow can be altered to avoid the problem. Compensating an activity A involves executing a *compensation subroutine* that attempts to undo the effects of the previous execution of A. This compensation can be very expensive, so it is important to minimize compensation scope to avoid unnecessary compensation effort. In this paper, those activities that require compensation are called the *compensation set* [9] of a failure.

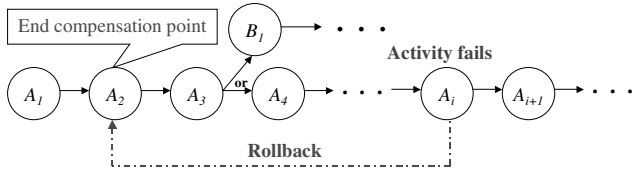

Fig. 1. The basic failure-recovery model

The complex failure-recovery process is illustrated in Fig. 1. We assume that the execution sequence of the activities is A_1, A_2, A_3, …, and A_i in that order; and that a failure occurred in activity A_i. The rollback of workflow process returns the execution to A_2, i.e., A_2 is the ECP of this failure. The workflow engine then compensates those activities in A_3–A_i that are in the compensation set. If an activity needs to be compensated, then the corresponding compensation subroutine of the occurred failure is executed. After all the compensations are finished, i.e., all the activities in the compensation set are compensated, the workflow process restarts from A_2. In the re-execution, the workflow process may proceed either from B_1 or A_4 after the end of A_3.

The failure-recovery models proposed in [9-11] have some drawbacks. First, they only allow specification of the ECP and compensation set of a failure in an activity in a static way before the workflow process is compiled. This limits the flexibility of failure recovery. A more flexible way is to compute the ECP and compensation dur-

ing the process run-time according to the execution results of activities. Second, the ECP and compensation set are specified by explicitly using the names (or some kind of identities) of activities. Therefore, inserting or deleting activities to or from a workflow process may require modification of the failure-recovery specification in another activity. Although Du et al. [9] proposed a compensation scoping strategy that allows the workflow designer to specify some designated compensation scopes based on data dependencies between workflow activities, their model does not allow the workflow designer to specify the ECP and compensation set in a flexible way, such as by using a programming language. In this paper, we propose a new language, called the workflow failure-handling (WfFH) language, which supports a workflow failure-recovery model with the following features:

1. The ECP and compensation set can be computed or derived during the workflow process run-time according to the execution results and status of workflow activities, as well as the type of failures.
2. The workflow designer can use the WfFH language to program the computation of the ECP and compensation. With the WfFM codes written by the workflow designer, the WfMS employs data-flow analysis technology to compute the ECP and compensation when the failure occurs.
3. The definition of failure recovery and compensation scope between activities can be absolutely independent. With this, inserting or deleting activities to or from a workflow process may not require modification of the failure-recovery specification in other activities. This reduces the maintenance overhead of workflow process dramatically and increases the reusability of workflow activities.
4. The details of information related to the failure can be sent to the ECP and activities in the compensation set to activate the most appropriate compensation for the failure situation that has occurred.
5. There can be multiple failures in a single activity, and each such failure can activate different failure-recovery processes.

The remainder of the paper is organized as follows. Section 2 explains, with the aid of some examples, why a language like the WfFH language is needed in WfMSs. Section 3 presents the WfFH language we propose. In Section 4, we detail the system architecture and implementation for a WfMS to support the WfFH language. Section 5 presents the experimental result. Section 6 concludes the paper.

2 Motivation Examples

In this section, we will begin with a simplified example of a workflow process to explain why the WfFH language is needed in the failure recovery of the WfMS (see Fig. 2). Assume that the workflow process is a selling process of a wholesale merchant. It consists of nine activities, A_1, A_2, …, and A_9 as well as five failures F_1, F_2, …, and F_5:

- Activity A_1. Receive order.
- Activity A_2. Build up or update customer information.

- Activity A_3 (named *input_order*). Input the order details.
- Activity A_4. Check the customer transaction and credit records.
- Activity A_5. Check the inventory records.
- Activity A_6. (named *evaluate_order*) Evaluate if the order should be accepted.
- Activity A_7. Pick up goods from the warehouse according to the order specified, and then deliver the goods to the customer.
- Activity A_8. Send the rejection letter to customer.
- Activity A_9. Bill customer.
- Failure F_1. Inventory-insufficient.
- Failure F_2. Over-quantity.
- Failure F_3. Goods were not successfully delivered.
- Failure F_4. Goods were rejected by customer because they did not conform to the order.
- Failure F_5. Unable to get the payment for goods.

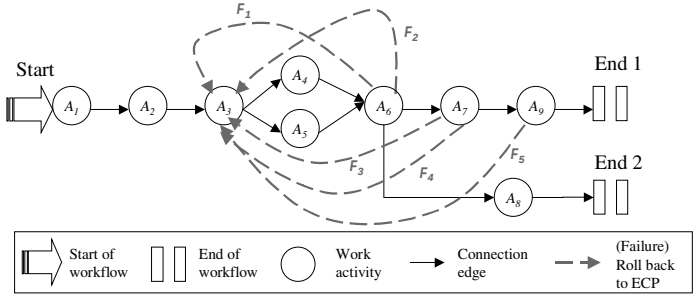

Fig. 2. An example of a workflow process

Below we describe some cases where the static method of specifying the ECP and compensation set could cause some problems in the design and maintenance of the workflow process.

The first problem is that inserting or deleting an activity may result in a change being required to the specification of the ECP and the compensation set in other activities of the workflow process. For example, assume that A_6 should rollback to the earliest activity that relates to the ordered goods when activity A_6 causes an *inventory-insufficient* failure. In this case the workflow designer specifies the ECP and compensation set to be A_3 and $\{A_3, A_6\}$, respectively. However, after that, if the workflow designer has to change the workflow process by adding an activity A_{10} named *confirm_order*, which is phone confirmation of the order after A_2 (see Fig. 3), the workflow designer has to change the ECP and compensation set of all the five failures F_1, $F2$, …, and F_5. For the setting of each failure, its ECP and compensation set may have to be modified, e.g., the ECP and compensation set of inventory-insufficient failure of A_6 should be changed to A_{10} and $\{A_{10}, A_3, A_6\}$, respectively. This kind of maintenance overhead for failure-recovery specification can be even more serious if a subflow is added to an existing workflow process. This is called composite workflow (or nested workflow) [16]; see Fig. 4A. In this case, specification of the ECP and compensation set for the inventory-insufficient failure of A_6 should consider all the activities in the added subflow – this can be very tedious and error-prone. We con-

sider that a good way to reduce the maintenance overhead is to provide a mechanism to let the workflow designer change the flow of the workflow process without having to change the failure-recovery definitions of other activities. The failure-recovery mechanism in [9–11] cannot solve this problem.

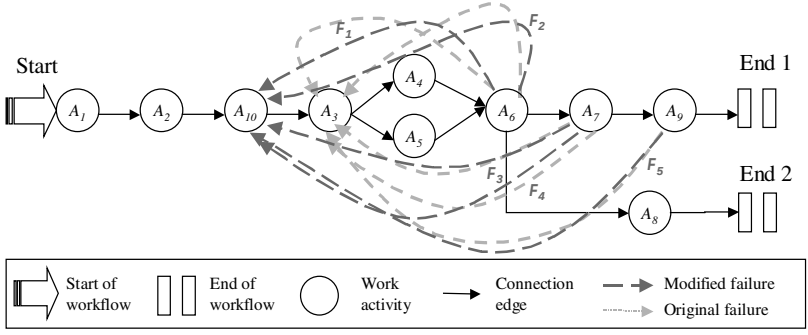

Fig. 3. Change of failure settings after inserting a new activity

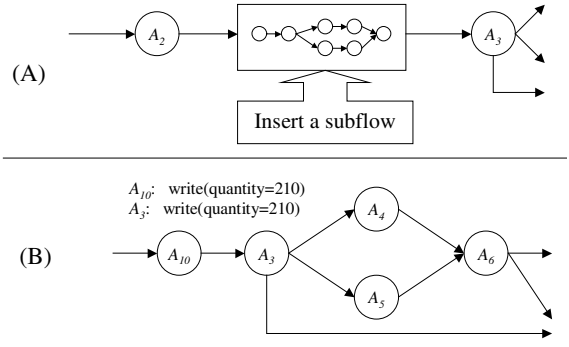

Fig. 4. Problems in changing the workflow process

The second problem is that it may not be possible to determine the ECP or compensation set of a failure until the run-time of the workflow process. This situation is illustrated in Fig. 4B. Assume that the wholesale merchant sets the quantity limitation for some goods during a peak period to, say, 100. Thus, A_6 may cause another failure called *over-quantity* for some specific goods. However, the quantity of the ordered good may be set either by activity A_{10} or by activity A_3, depending on the execution of A_{10} and A_3: if A_{10} does not set it, then A_3 must do so. Thus, the activity that sets the quantity can only be obtained according to the execution results of activities A_{10} and A_3. That is, the ECP or the compensation for over-quantity failure of A_6 depends on the execution results of previously executed activities. The WfMS should therefore compute or derive the ECP and compensation set according to the execution results of activities after the occurrence of the failure. The failure-recovery mechanism in [9–11] again cannot solve this problem.

The third case is that a single activity may cause multiple different failures. For example, activity A_6 may cause inventory-insufficient and over-quantity failure. The ECPs and compensation sets of these failures are all different. The failure-recovery

models proposed in [9–11] do not mention how to implement a WfMS which allows multiple failures to occur in a single activity: here we propose an architecture of the WfMS to support multiple failures in a single activity.

The activities of a workflow process usually involve arbitrarily complex functions. Thus, the activities can be very expensive to compensate and re-execute. It is therefore very important to decide the most-recent ECP and the minimal compensation set during the failure-recovery process. As we have discussed with the example shown in Fig. 2, the static way of specifying the ECP and compensation set causes difficulties in maintaining the failure-recovery definition when the workflow process is changed. Also, it cannot use run-time results of activities to compute the ECP and compensation set. Thus, we have designed the WfFH language to solve these problems.

3 The WfFH Language

Before we introduce the WfFH language, we first present the architecture of the workflow process definition which embeds the WfFH programs to support flexible and data-flow-analysis-based failure recovery. The WfFH program is embedded in the definition of the activity. The workflow process definition consists of at least the definitions of all of its activities as well as the specification of the control and data flows, including conditional branching, concurrent execution of activities, looping, and other complex control structures. Fig. 5 illustrates the basic architecture of the definition of workflow process and activity. The flow of activities is presented as a directed graph that defines the execution order of the activities in the process. Nodes and lines in the graph represent activities and control flow, respectively. Note that the directed graph is not the unique way of representing the flow of the activities; existing methods of specifying the flow of activities include the Petri Net [12,13] and generalized process-structure grammar [14].

3.1 The Structure of Activities to Support WfFH

An activity is a logic step within a workflow. The activity definition is also shown in Fig. 5, which contains basic information for activity execution and additional failure-recovery definitions. The basic information for activity execution includes at least the starting and stopping conditions, the users who can participate, the tools and/or data needed to complete this activity, the constraints on how the activity should be completed (such as the time limits), and the *execution codes* of the activity. The additional failure-recovery definition in the activity for data-flow-analysis-based failure recovery comprises *recovery definitions* and definitions of the *compensation subroutines*. The execution code is the program that executes the activity and which may trigger the failure-recovery process when the execution causes a failure.

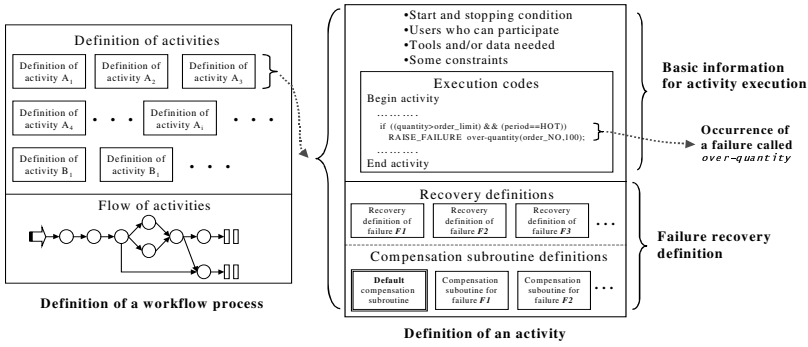

Fig. 5. The definition of the workflow process and activities

3.2 The Skeleton of the Recovery Definition

In this paper, we propose using the WfFH language to specify the recovery definition. When a failure occurs, the workflow engine executes the WfFH codes of the corresponding recovery definition of the failure to compute the ECP and compensation set. All the activities in the compensation set should be compensated. In the compensation of an activity, the workflow engine first checks if there is a corresponding compensation subroutine of the occurred failure in the activity – this compensation subroutine will be executed if it exists; otherwise the *default* compensation subroutine will be executed. Since an activity may cause different failures, the activity definition may contain multiple recovery definitions and compensation subroutines. According to the syntax of WfFH language, each recovery definition has the form shown in Fig. 6.

```
       FAILURE Name(type1 arg1, type2 arg2, …)
1.1     BEGIN Declaration
1.2         // make declaration of variables
1.3         Activity_set a_set1, a_set2;
1.4         Resource R1, R2;
                    …
1.5     END Declaration
2.1     BEGIN Compensation Set Computation
2.2       // code to derive compensation set here
                  …
2.3       DO_Compensation a_set1;
2.4     END Compensation Set Computation
3.1     BEGIN ECP Computation
3.2       // code to make ECP computation set here
                    …
3.3       Rollback_To a_set2;
3.4     END ECP Computation
       END Failure
```

Fig. 6. The skeleton of a recovery definition written in the WfFH language

A recovery definition programmed with the WfFH language consists of a header and a body. The header begins with a keyword "FAILURE," which is followed by the name of the failure and a sequence of arguments. The syntax of the argument list of the WfFH language is the same as that for Java [15]. These arguments have two uses:

(1) to provide information for the computation of the ECP and compensation set, and (2) they will be passed to the compensation subroutines of the compensated activities. The compensation subroutines could use the received arguments to perform the most appropriate compensation. The body comes after the header, and comprises three sections. The first section in the body, the *declaration section*, includes declarations of variables used in the next two sections. Variables declared in the declaration section include:

- Variables used to store set of activities. The compensation set and ECP of a failure consist of set of activities. We can use the keyword "Activity_set" to make the declaration (see line 1.3 of Fig. 6).
- Variables used to store resources. For data-flow-analysis-based specification and computation of the compensation set and ECP, we have to declare some variables which store shared resources accessed by different activities (see line 1.4 of Fig. 6).
- Others. The WfFH language allows the use of all the data types defined in Java [15], including types defined by the user.

3.3 Methods in WfFH

The data-flow analysis technology (or global data-flow analysis technology) for compiler optimization analyzes how data values are modified across basic blocks of program statements [17]. The data-flow analysis technology used in our failure-recovery model is derived from the data-flow analysis technology used in compiler optimization. However, instead of analyzing the data values modified in program statements, our model analyzes the resource accesses of activities in workflow processes. To build up a data-flow-analysis-based failure-recovery mechanism, i.e., to compute the ECP and compensation set using data-flow analysis technology, the workflow engine which controls the execution of the activities has to monitor and record all the accesses to the *shared resources* of each activity. During the execution of a workflow activity, it may access private and/or shared resources. Private resources are defined as data used only within the execution of a single activity, whereas shared resources are data that are created, read, written, or modified by multiple activities, or data which are considered as the output results of the workflow process. In the current definition and implementation of the WfFH language, they are two kinds of resources that the activity can access: *files* and *database records*. The accesses to shared resources issued by activities are stored in an *execution-log database*. Each access record stored in the execution-log database contains at least the activity which performs the access, the time and type of the access (i.e., insert, delete, update, or create), and the file name or the primary keys (or table name) of the access database records. File accesses are easy to monitor, but monitoring accesses to database records is much more difficult. We do this using execution-monitoring technology which is proposed and developed for reachability testing of client–server database applications [18,19]. The next section in the definition is the *compensation-set computation* section, which the WfMS executes to derive the compensation set of this failure. Its syntax is similar to Java, with some extension of functions to support the computation of compensation sets. They can be classified into three main groups as described below.

The first group comprised functions which return the set of activities. We call them activity-manipulation functions:

- ACTIVITY("*name_of_activity*"). Given the name of an activity in the workflow process as the argument of this function (possibly containing the wildcard operator * or ?), this function returns the specified activities. For example, *a_set1=ACTIVITY("check*")* will return the set of activities whose names are prefixed with "*check*."
- ACTIVITY(*ALL*). This function returns the set of all the activities in the workflow process.
- ACTIVITY(*EXECUTED*). This function returns the set of all the activities which have been executed.
- ACTIVITY(*THIS*). This function returns the set of the activity which causes the failure (the current activity).
- ACTIVITY_READ(*resource*). This function returns the set of all the activities which performed a read operation on *resource*.
- ACTIVITY_WROTE(*resource*). This function returns the set of all the activities which performed a write operation on *resource*.
- ACTIVITY_READ_WROTE(*resource*). This function returns the set of all the activities which performed read and write operation on *resource*.
- ACTIVITY_DELETE(*resource*). This function returns the set of all the activities which performed the delete operation on *resource*.
- ACTIVITY_CREATE(*resource*). This function returns set of all the activities which created *resource*.
- FIRST(*activity_set*). This function returns the earliest executed activity in the *activity_set*.
- LAST(*activity_set*). This function returns the most-recent executed activity in the *activity_set*.

The second group comprises functions which return a set of resources. We call them resource-manipulation functions:

- SELECT_DB_RECORDS("*DB_Name*", "*select_sql_transaction*"). This returns the database records which are queries with the transaction *select_sql_transaction* from the database *DB_Name*.
- FILES(f_1, f_2, …, f_n). This returns the files f_1, f_2, …, f_n as a resource type.
- RESOURCE_READ_BY(*activity_set*). This returns all the resources read by *activity_set*.
- RESOURCE_WROTE_BY(*activity_set*). This returns all the resources written by *activity_set*.
- RESOURCE_DELETED_BY(*activity_set*). This returns all the resources deleted by *activity_set*.
- RESOURCE_CREATED_BY(*activity_set*). This returns all the resources created by *activity_set*.

The functions SELECT_DB_RECORDS and FILES return the specified resources from the shared resources of the workflow process. RESOURCE_READ_BY,

`RESOURCE_WROTE_BY`, `RESOURCE_DELETED_BY`, and `RESOURCE_CREATED_BY` allow the user to determine what resources are accessed by activities.

The third group comprises the set of operations which support the activity- and resource-manipulation functions:

- `INTERSECT(S1, S2)`. This returns the intersection of activity sets or resource sets (i.e., `{x | x ∈ S1 and x ∈ S2}`).
- `UNION(S1, S2)`. This returns the union of activity sets or resource sets (i.e., `{x | x ∈ S1 or x ∈ S2}`).
- `SIZE_OF(S)`. This returns the number of elements in set S corresponding either to the set of activities or resources.
- `DIFFERENCE(S1, S2)`. This returns the set which deletes element of S_2 from S_1 (i.e., `{x | x ∈ S1 and x ∉ S2}`).
- `S(i)`. This returns the i-th element in set `S`.

The codes in this section also conform with the statement and loop structures of Java, which allows the recovery definitions programmed with the WfFH language to be easily translated into Java objects. A statement specifying the resulting compensation set should appear at the end of this section, consisting of a keyword "`DO_Compensation`" and an activity set variable which stores the resulting compensation set (see line 2.3 of Fig. 6).

The final section in the recovery definition for the WfFH language is the *ECP computation* section. The programmer uses this to specify how to compute the ECP. The syntax and available functions of this section are identical to those for the compensation-set computation section described above. Furthermore, it inherits the computation results of all the variables used in the section in which the compensation set is computed. The ECP computation section also ends with a statement to specify the resulting ECP. It begins with a keyword "Rollback_To" which follows an activity-set variable which stores the resulting ECP.

We now provide several examples to illustrate the WfFH language. The WfFH Code in Fig. 7A demonstrates how to use the WfFH language to specify the inventory-insufficient failure of the motivating example we presented in Section 2. We set the compensation set to be activities whose names are postfixed with *_order*, because the names of A_3, A_6, and A_{10} are *input_order*, *evaluate_order* and *confirm_order*, respectively. It is obvious that there is no need to change the recovery definition of inventory-insufficient failure of A_6 when A_{10} is added to the workflow process, and so we did not employ the data-flow analysis technology of the WfFH language in it. However, in the definition of the over-quantity failure (see WfFH Code Example in Fig. 7B), we have to use data-flow analysis technology in the computation of ECP and compensation set. There are two arguments: *order_No* and *order_Limit*. The *order_No* argument is used in the computation of ECP and compensation set, and both of them will be sent to the compensation subroutines. The database record of "*quantity*" is first retrieved and stored in "*OQuantity*." This database record refers to the quantity of the order stored by either A_{10}or A_6. Then, each executed activity is checked to see if it wrote to "*OQuantity*," and all those that did are added to the compensation set. The ECP is the earliest activity in the compensation set.

(A)

```
FAILURE inventory-insufficient( )
  BEGIN Declaration
    Activity_Set a_set1, a_set2, a_set3;
    Activity_Set comp_set, ecp;
  END Declaration
  BEGIN Compensation Set Computation
    a_set1 = ACTIVITY(EXECUTED);
    a_set2 = ACTIVITY("*_order");
    comp_set = INTERSECT(a_set1, a_set2);
    DO_Compensation comp_set;
  END Compensation Set Computation
  BEGIN ECP Compution
    ecp = FIRST(comp_set);
    Rollback_To ecp;
  END ECP Compution
END FAILURE
```

(B)

```
FAILURE over-quantity(int order_No, int order_Limit)
  BEGIN Declaration
    Activity_Set a_set1;
    Activity_Set comp_set, ecp;
    Resource OInventory;
    int i, size;
  END DECLARATION
  BEGIN Compensation Set Computation
    OQuantity = SELECT_DBRECORDS ("WF_DB1", "SELECT quantity
        FROM OrderForm WHERE OrderForm.OrderNo=" + Order_No);
    a_set1 = ACTIVITY(EXECUTED);
    size = SIZE_OF(a_set1);
    for (i = 1; i <= size ; i++)
    {
      if (INTERSECT(RESOURCE_WROTE_BY(a_set1(i), OQuantity) !=
          EmptySet)
            comp_set = UNION(comp_set, a_set1(i));
    }
    DO_Compensation comp_set;
  END Compensation Set Computation
  BEGIN ECP Compution
    ecp = FIRST(comp_set);
    Rollback_To ecp;
  END ECP Compution
END FAILURE
```

Fig. 7. Two WfFH code examples.

4 The System Architecture and Implementation

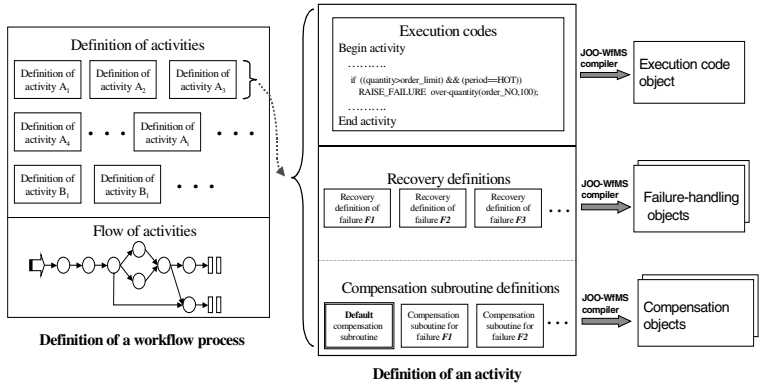

Fig. 8. Generating Java object codes of a workflow process

Here we are implementing a Java-based object-oriented WfMS (JOO-WfMS [21]) which supports the WfFH language. Fig. 8 shows the workflow process definition which conforms to the syntax of the JOO-WfMS as compiled using the *JOO-WfMS compiler* to generate Java object codes (or Java class files), including some activity objects, a flow object, some failure-handling objects, and some compensation objects. An activity object includes information and codes for executing the activity. Also, each activity object is associated with some failure-handling objects and compensation objects. The flow object is used to control the order of execution of the activities. These Java objects are sent to the JOO workflow engine which activates and controls the execution of the workflow process encoding by these Java objects. There are stan-

dard Java interface definitions for workflow, flow control, activity, execution code, failure, and compensation handler objects in JOO-WfMS.

The way to activate the failure-recovery process is specified in the execution code of the definition of the activity (see Fig. 5). In the JOO-WfMS, the execution code of an activity is also a Java program with the *failure-recovery extension*, which is of the following form:

```
RAISE_FAILURE Failure_Name(arg1, arg2, …,argn);
```

The JOO-WfMS compiler translates the above statement into the following Java program:

```
//To generate an array args to store arg1, arg2, …, and argn
Object[] args= new Object[n];
args[0]= arg1;
args[1]=arg2 ;
   …
args[n]=argn ;
// To instantiate a failure object
Failure fail = activity.getFailure(Failure_Name) ;
//To  setup the failure arguments
fail.SetArguments(args) ;
// To throw a Java exception to start the failure recovery process
throw new RaiseFailureException(fail) ;
```

The following is a fragment from the execution code of the activity definition of A_6. It specifies how to activate recovery process of the *inventory_insufficient* and *over-quantity* failures:

```
…
if (inventory < ordered_quantity)
   RAISE_FAILURE inventory-insufficient( );
if ((quantity > order_limit) && (period==HOT))
   RAISE_FAILURE over-quantity(order_No, order_limit);
…
```

The "`inventory`," "`ordered_quantity`," "`quantity`," "`order_limit`," "`period`," and "`order_No`" are local variables in the execution codes of A_6. If the value of "`inventory`" is less than "`ordered_quantity`," it will activate the failure recovery according to the recovery definition named "*inventory-insufficient*" (see Fig. 7). Also, if "`quantity`" is greater than "`order_limit`" and the "`period`" is "`HOT`," the recovery for an over-quantity failure will start. The code translated by JOO-WfMS compiler is shown as follows:

```
if (inventory < ordered_quantity) {
  Failure fail =activity.getFailure("inventory-insufficient") ;
  throw new RaiseFailureException(fail) ;
}
if ((quantity > order_limit) && (period==HOT)) {
  Object[] args= new Object[2] ;
  args[0] = order_No ;
  args[1] = order_limit ;
Failure fail =activity.getFailure("over-quantity") ;
fail.SetArguments(args) ;
   throw new RaiseFailureException(fail) ;
}
```

Fig. 9 uses an example to illustrate how the JOO-WfMS compiler translates the WfFH code in Fig. 7B into a Java program. We currently have a Java implementation

of the JOO-WfMS that supports our proposed workflow failure-handling model. For details about it, refer to [21].

Over-Quantity failure in WfFH

```
FAILURE over-quantity (int order_No, int order_Limit)

//---------Variable declarations---------------------
  BEGIN Declaration
    Activity_Set a_set1;
    Activity_Set comp_set, ecp;
    Resource Oinventory, OQuantity;
    int i, size;
  END DECLARATION

//-----------------Compute compensation set ---------
  BEGIN Compensation Set Computation
    OQuantity=SELECT_DBRECORDS("WF_DB1", "SELECT
     quantity FROM OrderForm WHERE
     OrderForm.OrderNo=" + Order_No);
    a_set1 = ACTIVITY(EXECUTED);
    size = SIZE_OF(a_set1);
    for (i = 1; i <= size ; i++)
    {
      if (INTERSECT(RESOURCE_WROTE_BY(a_set1(i),
         OQuantity) != EmptySet)
             comp_set = UNION(comp_set, a_set1(i));
     }
     DO_Compensation comp_set;
  END Compensation Set Computation

//--------------Compute  ECP -----------------------
  BEGIN ECP Compution
    ecp = FIRST(comp_set);
    Rollback_To ecp;
  END ECP Compution
END FAILURE
```

Java Code

```
public class Failure_Over_Quantity extends Failure
{
 //---------Variable declarations---------------------
       ActivitySet a_set1 ;
       ActivitySet comp_set, ecp ;
       ResourceSet Oinventory, OQuantity;
       int i,size;

 //----------Methods to hanlde arguments of failure----
       public void SetArguments( Object[] args ){
         for( int i = 0 ; i < args.length ; i++ )
             failureArgs.addElement( args[i] );
       }
       public Object[] GetArguments(){
         Object[] obj = new Object
                  [ failureArgs.size() ];
          failureArgs.copyInto( obj );
          return obj;
       }
       public Class[] GetArgumentTypes(){
         Class[] cla = new Class
                  [ failureArgs.size() ];
         for( int i = 0 ; i < failureArgs.size() ; i++)
           cla[i] = failureArgs.elementAt(i).getClass();
         return cla;
       }
 //-----------------Compute compensation set ----------
   public  ActivitySet CompensationSetComputation() {
     comp_set = new ActivitySet () ;
      int Order_No =
      ((Integer)GetArguments()[0]).intValue();
      OQuantity=SELECT_DBRECORDS("WF_DB1",
          "SELECT quantity FROM OrderForm WHERE
          OrderForm.OrderNo=" + Order_No);
      a_set1=ACTIVITY_EXECUTED(
      activity.GetBelongWorkflow().GetWorkflowID());
      size=SIZE_OF(a_set1) ;
      for (i=1,i<=size;i++)
      {
         if (INTERSECT(RESOURCE_WROTE_BY(a_set1(i),
               OQuantity) != EmptySet)
                comp_set = UNION(comp_set, a_set1(i));
       }
      return comp_set ;
    }

 //--------------Compute  ECP ------------------------
   public ECP ECPComputation() {
     ecp=FIRST( comp_set ) );
     return new ECP(ecp);
    }
}
```

Fig. 9. Translating of the WfFH code into the Java program.

5 Performance Evaluation

We conduct a series of experiments to evaluate the performance of the failure-handling system in JOO-WfMS. The performance was evaluated by measuring the time required to instantiate the execute code, compensation, and failure-handling objects as well as the computation of compensation set and ECP. All the experiments

were performed on a PC with a 1-GHz Pentium VI processor, 512 MB of RAM, the MS Windows 2000 operating system, and Java Development Kit 1.4.1_01 [22].

Table 1. The times required to instantiate the execute code, compensation handler, and failure objects (in seconds)

Activity A_1		
Execution code object (code_a1.ser – 169 bytes)	0.025▲	0.021◆
Activity A_2		
Execution code (code_a2.ser – 169 bytes)	0.025▲	0.017◆
Activity A_3		
Execution code (code_a3.ser – 171 bytes)	0.022▲	0.015◆
Compensation handler object of failure F_1 (cfii_a3.ser – 531 bytes)	0.029▲	0.019◆
Compensation handler object of failure F_2 (cfoq_a3.ser – 531 bytes)	0.029▲	0.020◆
Compensation handler object of failure F_3 (cfwc_a3.ser – 531 bytes)	0.033▲	0.020◆
Compensation handler object of failure F_4 (cfwo_a3.ser – 531 bytes)	0.025▲	0.020◆
Compensation handler object of failure F_5(cfib_a3.ser – 531 bytes)	0.031▲	0.019◆
Activity A_4		
Execution code (code_a4.ser – 169 bytes)	0.022▲	0.016◆
Activity A_5		
Execution code (code_a5.ser – 169 bytes)	0.022▲	0.016◆
Activity A_6		
Execution code (code_a6.ser – 169 bytes)	0.024▲	0.016◆
Failure object of failure F_1 (fii.ser – 654 bytes)	0.030▲	0.020◆
Failure object failure F_2 (foq.ser - 698 bytes)	0.032▲	0.020◆
Activity A_7		
Execution code object (code_a7.ser – 169 bytes)	0.021▲	0.015◆
Failure object of failure F_3 (fwc.ser – 653 bytes)	0.029▲	0.020◆
Failure object of failure F_4 (fwo.ser – 651 bytes)	0.030▲	0.020◆
Activity A_8		
Execution code object (code_a8.ser – 169 bytes)	0.021▲	0.015◆
Activity A_9		
Execution code object (code_a9.ser – 169 bytes)	0.021▲	0.015◆
Failure object of failure F_5 (fib.ser – 651 bytes)	0.030▲	0.020◆
Activity A_{10}		
Execution code (code_a10.ser – 170 bytes)	0.023▲	0.016◆
Compensation handler object of failure F_1 (cfii_a10.ser – 532 bytes)	0.028▲	0.022◆
Compensation handler object of failure F_2 (cfoq_a10.ser – 532 bytes)	0.027▲	0.024◆
Compensation handler object of failure F_3 (cfwc_a10.ser – 532 bytes)	0.026▲	0.022◆
Compensation handler object of failure F_4(cfwo_a10.ser – 532 bytes)	0.027▲	0.019◆
Compensation handler object of failure F_5(cfib_a10.ser – 532 bytes)	0.027▲	0.022◆

▲ The serialized Java objects are on the local hard disk.
◆ The serialized Java objects are on a remote Web site in the same network segment (100Mbps).

Table 1 presents times required to instantiate the execute code, compensation, and failure-handling objects of the workflow shown in Fig. 3. In the JOO-WfMS, all these objects are translated into serialized Java objects to support dynamic linking and execution. We also list the size of the serialized Java objects, e.g., the execution code object of activity A_1 comprised 169 bytes. The serialized Java objects are either on the

local hard disk or on a remote Web site in the same network segment (100Mbps). Table 2 shows times required to compute the compensation set and ECP. Experimental results obtained demonstrate the efficiency and practicability of the proposal failure-handling model.

Table 2. The times required to compute the compensation set and ECP (in seconds)

Failure F_1	3.20E-4
Failure F_2	6.26E-4
Failure F_3	1.88E-4
Failure F_4	9.40E-5
Failure F_5	9.40E-5

6 Conclusion

The paper has addressed workflow failure recovery in WfMSs. The new failure-recovery model proposed in this paper provides the workflow designer with a WfFH language in which to write programs to guide the recovery. New features of the proposed new model include: (1) computing the ECP and compensation set according to the execution results during the workflow process run-time, (2) employing data-flow analysis technology to guide the failure-recovery process, (3) allowing multiple failures to occur in a single activity with different recovery definitions, and (4) minimal dependency between activities of the failure-recovery definitions, thereby reducing the maintenance cost and improving the reusability of the workflow process.

Due to space limitations, some important issues are discussed only briefly or not at all. First, in our implement, each recovery definition written in the WfFH language is translated into a Java object, which makes the JOO-WfMS use a Java-based workflow engine to activate the failure-recovery process. Second, we have also designed an activity-monitoring protocol for recording the accesses of shared resources, and this forms the basis for the data-flow analysis technology used in our model. Third, we have not described how to send the arguments of the recovery definitions to the compensation subroutines. Refer to [21] for more details.

References

1. D. Georgakopoulos, M. Hornick, and A. Shet. Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. Distributed and Parallel Databases, Vol. 3, No. 2, 1995, Pages 119–153.
2. Shi Meilin, Yang Guangxin, Xiang Yong, and Wu Shangguang. Workflow Management Systems: A Survery. International Conference on Communication Technology, 1998.
3. A. Elmagarmid, and W. Du. Workflow Management: State of the Art vs. State of the Market. Proceedings of NATO Advanced Study Institute on Workflow Management Systems, 1997.
4. Workflow Management Coalition. Workflow Reference Model. *Workflow Management Coalition Standard*, WfMC-TC-1003, 1994.
5. Workflow Management Coalition. Workflow Management Systems: A Survery. *Workflow Handbook*, 2001.

6. Nina Edelweiss and Mariano Nicolao. Workflow modeling: Exception and Failure Handling Rrepresentation. *IEEE International Conference of the Chilean Computer Science Society*, 1998.
7. Fabio Casati, Stefano Ceri, Stefano Paraboschi, and Guiseppe Pozzi. Specification and Implementation of Exceptions in Workflow Management Systems. *ACM Transactions on Database Systems*, Vol. 24, No. 3, September 1999, Pages 405–451.
8. Claus Hagen and Gustavo Alonso. Exception Handling in Workflow Management Systems. *IEEE Transactions on Software Engineering*, Vol. 26, No. 10, October 2000, Pages 943–958.
9. Weimin Du, Jim Davis, and Ming-Chien Shan. *Flexible Specification of Workflow Compensation Scopes*. ACM Group, Phoenix, Arizona, USA, 1997.
10. M. Kamath and K. Ramamrithan. Failure Handling and Coordinated Execution of Concurrent Workflows. *IEEE International Council for Open and Distance Education*, 1998.
11. J. Eder and W. Liebhart. Workflow recovery. *IEEE International Conference on Cooperative Information Systems*, 1996.
12. W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, Vol. 8, No. 1, 1998, Pages 21–66.
13. Sea Ling and H. Schmidt. Time Petri nets for workflow modelling and analysis. *IEEE International Conference on Systems, Man, and Cybernetics*, 2000.
14. N.S. Glance, D.S. Pagani, and R. Pareschi. Generalized process structure grammars (GPSG) for flexible representations of work. *Proceedings of Conference on Computer Supported Cooperative Work*, 1996.
15. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification (First Edition)*. Addison-Wesley, Reading, Massachusetts, USA, 1996.
16. D. Worah and Amit Sheth. Transactions in Transactional Workflows. In: S. Jajodia and L. Kerschberg (eds) *Advanced Transaction Models and Architectures*, Kluwer Academic, Boston, Massachusetts, USA, 1997.
17. Afred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, USA, 1986.
18. Gwan-Hwan Hwang, Huey-Der Chu, and K.C. Tai. Testing of Non-Deterministic Client–Server Database Applications. *The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, June 25–28, 2001, Monte Carlo Resort, Las Vegas, Nevada, USA.
19. Gwan Hwan Hwang, Sheng-Jen Chang, and Huey-Der Chu, "*Testing Client/Server Database Applications,*" Technical Report, National Taiwan Normal University, 2002. http://bashful.ice.ntnu.edu.tw/~ghhwang/papers/Testing_CSDB.pdf.
20. Sun Microsystem, Inc. JSR-000053 Java™ Servlet 2.3 and JavaServer Pages™ 1.2 Specifications. *http://jcp.org/aboutJava/communityprocess/first/jsr053/index.html*, March 2002.
21. Gwan-Hwan Hwang and Yung-Chuan Lee, *"The Architecture of JOO-WfMS and its implementation,"* Technical Report, National Taiwan Normal University, 2003.
22. Sun Microsystem, *"The Source for Java(TM) Technology,"* http://java.sun.com, 2002.