A Workflow Specification Language and its Scheduler *

Nesime Tatbul, Sena Arpinar, Pinar Karagoz, İbrahim Cingil,
Esin Gokkoca, Mehmet Altinel, Pinar Koksal, Asuman Dogac
Software Research and Development Center
Dept. of Computer Engineering
Middle East Technical University (METU)
06531 Ankara Turkiye
asuman@srdc.metu.edu.tr
Tamer Ozsu
Department of Computing Science
University of Alberta, Edmonton, Alberta
Canada T6G 2H1
ozsu@cs.ualberta.ca

Abstract

This paper describes a workflow specification language, namely MFDL, and the implementation of its scheduler in a distributed environment. Distributed nature of the scheduling provides failure resilience and increased performance. Since workflow scheduling and management is highly affected from the way the workflow is specified, a workflow specification language should be efficient to prevent the problems of complexity in workflow specification and difficulties in debugging/testing the further steps of the workflow management system development. MFDL, being a block-structured procedural workflow specification language, is capable of defining a workflow in an easy, comprehensible and clear way so that implementation of the scheduler is simplified. The paper also presents task handling in the system through a CORBA compliant ORB.

1 Introduction

Workflow management systems (WFMSs) automate the execution of business processes. WFMSs achieve considerable improvements in critical, contemporary measures of performance, such as cost, quality, service, and speed by coordinating and streamlining complex business processes within large organizations.

A workflow system can be defined as a collection of processing steps (also termed as tasks or activities) organized to accomplish some business process. A task can be performed by one or more software systems, or, by a person or a team, or a combination of these. In addition to the collection of tasks, a workflow defines the order of task invocation or condition(s) under which tasks must be invoked (i.e. control-flow) and data-flow between these tasks.

In general a workflow task is considered to be a black box that is functional in nature, i.e., the functionality of the task is orthogonal to that of the workflow process. The tasks could be transactional or non-transactional in nature. Transactional tasks are those that access data controlled by Resource Managers (RMs) with transactional properties (i.e. ACID). Non-transactional tasks access data controlled by RMs without transactional properties such as file systems.

In a workflow specification language, the tasks involved in a business process and the execution and data dependencies between these tasks are provided. One of the weaknesses of current WFMSs is their specification language (Sheth et.al. 1997): state-of-the-art workflow specification languages are unstructured and/or rule based. Unstructured specification languages make debugging/testing of complex workflow difficult and rule based languages become inefficient when they are used for specification of large and complex workflow processes. This is due to the large number of rules and overhead associated with rule invocation and management.

* This work is partially being supported by the Turkish State Planning Organization, Project Number: AFP-03-12DPT.95K120500, by the Scientific and Technical Research Council of Turkey, Project Number: EEEAG-Yazilim5, by NATO Collaborative Research Grant No: CRG.970108, and by Sevgi Holding (Turkey).

Furthermore, a workflow specification language that specifies the control-flow and data-flow among tasks by itself is not enough (Krishnakumar and Sheth 1995): there is a need for a task specification language to specify the task interface for executing the task at the processing entity, for handling the error messages produced by the processing entity and also for revealing the task state for control purposes.

In order to meet the requirements cited above, we have designed workflow definition language, namely, METUFlow Definition Language (MFDL) and implemented the scheduler corresponding to this specification. MFDL also has a graphical user interface developed through Java which makes it possible to define a workflow by accessing METUFlow through the Internet.

MFDL is a block-structured language that contains seven types of blocks, namely, serial, and_parallel, or_parallel, xor_parallel, contingency, conditional and iterative blocks. MFDL overcomes the problems of unstructured and rule based languages through its block structures. The further advantages brought by this language can be summarized as follows:

- A block structured language confines the intertask dependencies to a well formed structure which in turn proves extremely helpful in implementing a distributed scheduler for the system.
- A block clearly defines not only the data and control dependencies among tasks but also presents a well defined recovery semantics, i.e., when a block aborts, the tasks that are to be compensated and the order in which they are to be compensated are already provided by the block semantics.

Workflow systems are expected to work in distributed heterogeneous environments which are very common in enterprises of even moderate complexity. CORBA is one of the major standardization initiatives of the computer industry for handling heterogeneity in distributed environments (OMG 1991, Soley and Stone 1995). CORBA provides a standard communication mechanism which enables distributed objects to operate on each other. In CORBA only the ORB (Object Request Broker) knows the implementation details and actual locations of the components (objects) in the system. Clients and servers only know the interfaces of the components. The only way of communication is the requests and the responses. In this way a distributed, heterogeneous environment becomes virtually local and homogeneous to the client. The interfaces are defined using IDL (Interface Definition Language) which resembles a declarative subset of C++ but it is not a programming language, making ORB object development implementation language independent. METUFlow, by allowing CORBA-IDL to be used in task specification, makes it possible to invoke tasks in distributed heterogeneous environments and meets the need for a task specification languauge.

METUFlow has a distributed scheduling mechanism. In current commercial workflow systems, the workflow scheduler is a single centralized component. A distributed workflow scheduler on the other hand should contain several schedulers on different nodes of a network each executing parts of process instances. Such an architecture fits naturally to the distributed heterogeneous environments. Further advantages of such an architecture are failure resiliency and increased performance since a centralized scheduler is a potential bottleneck.

The paper is organized as follows: Section 2 describes the process model and its specification in METUFlow Definition Language. In Section 3, the distributed scheduling mechanism corresponding to MFDL is provided. Section 4 discusses the task handling in METUFlow. Finally, Section 5 concludes the paper.

2 The Process Model and MFDL

We define a workflow process as a collection of blocks, tasks and other subprocesses. A task is the simplest unit of execution. Processes and tasks have input and output parameters corresponding to workflow relevant data to communicate with other processes and tasks. We use the term activity to refer to a block, a task or a process. Blocks differ from tasks and processes in that they are conceptual activities which are present only to specify the ordering and the dependencies between activities.

The following definitions describe the semantics of the block types where A stands for an activity (block, task or process), B for a block and T for a task.

Serial Block B = $(A_1; A_2; A_3; .....; A_n)$: Start of a serial block B causes A_1 to start. Commitment of A_1 causes start of A_2 and commitment of A_2 causes start of A_3, and so on. Commitment of A_n causes commitment of B. If one of the activities aborts, the block aborts. If the block aborts, its committed activities should be compensated in the reverse order.

And_Parallel Block B = $(A_1$ & A_2 & ..... & $A_n)$: Start of an and_parallel block B causes start of all of the activities in the block in parallel. B commits only if all of the activities commit. If one of the activities aborts, the block aborts. If the block aborts, its committed activities should be compensated in parallel.

Or_Parallel Block B = $(A_1|A_2|.....|A_n)$: Start of an or_parallel block B causes start of all of the activities in the block in parallel. At least one of the activities should commit for B to commit but B can not commit until all of the activities terminate. B aborts if all the activities abort. If B aborts, its committed activities should be compensated in parallel.

Xor_Parallel Block B = $(A_1||A_2||.....||A_n)$: Start of an xor_parallel block B causes start of all tasks in the block in parallel. B commits if one of the activities commits, and commitment of one activity causes other activities to abort. If all of the activities abort, the block aborts.

Contingency Block B = $(A_1, A_2, ...A_n)$: Start of a contingency block B causes start of A_1. Abort of A_1 causes start of A_2 and abort of A_2 causes start of A_3,and so on. Commitment of any activity causes commitment of B. If the last activity A_n aborts, the block aborts.

Conditional Block B = (condition,A_1, A_2): Conditional block B has two activities and a condition. If the condition is true when B starts, then the first activity starts. Otherwise, the other activity starts. The commitment of the block is dependent on the commitment of the chosen activity. If the chosen activity aborts, then B aborts.

Iterative Block B = (condition;$A_1; A_2; .....; A_n$): The iterative block B is similar to serial block, but start of iterative block depends on the given condition as in a while loop and execution continues until either the condition becomes false or any of the activities aborts. If B starts and the condition is true, then A_1 starts and continues like serial block. If A_n commits, then the condition is reevaluated. If it is false, then B commits. If it is true, then A_1 starts executing again. If one of the activities aborts at any one of the iterations, B aborts. If B aborts, its committed activities for all the iterations should be compensated in the reverse order.

Compensation Activity of A = (A_c, AbortList(A_c)): The compensation activity A_c of A starts if A has committed and any of the activities in AbortList(A_c) has aborted. AbortList is a list computed in compile time which contains the activities whose aborts necessitate the compensation of A. If both an activity and its subactivities have compensation, only the compensation of the activity is used. If only the subactivities have compensation, it is necessary to use compensations of the subactivities to compensate the whole activity.

Undo Task of T = T_u: The undo task T_u of T starts if the non-transactional task T fails.

In addition to activities, there are also assignment statements in our language which access and update workflow relevant data.

We have implemented a specification language based on these structures, called METUFlow Definition Language (MFDL), within the scope of the METUFlow project. The following is an example workflow defined in MFDL:

```
TRANS_ACTIVITY register_patient (OUT int patient_id);
TRANS_ACTIVITY delete_patient(IN int patient_id);
USER_ACTIVITY  examine_patient (IN int patient_id,
      OUT int blood_test_type_list,
      OUT int roentgen_list)
      PARTICIPANT DOCTOR;
USER_ACTIVITY  blood_exam (IN int patient_id,
      IN int  blood_test_type_list, OUT STRING result)
      PARTICIPANT LABORANT;
USER_ACTIVITY  roentgen (IN int patient_id,
      IN int roentgen_list, OUT STRING result)
      PARTICIPANT ROENTGENOLOGIST;
USER_ACTIVITY  check_result (IN int patient_id,
      IN  string result1, IN STRING result2)
      PARTICIPANT DOCTOR;
USER_ACTIVITY  cash_pay (IN int patient_id)
      PARTICIPANT TELLER;
USER_ACTIVITY  credit_pay (IN int patient_id)
      PARTICIPANT TELLER;
DEFINE_PROCESS check_up (IN int patient_id)
{
  ACTIVITY register_patient register;
  ACTIVITY delete_patient delete;
  ACTIVITY examine_patient examine;
  ACTIVITY blood_exam blood;
  ACTIVITY roentgen roent;
  ACTIVITY check_result check;
  ACTIVITY cash_pay cash;
  ACTIVITY credit_pay credit;
  var STRING result1, result2;
  var int  blood_test_type_list, roentgen_list;
```

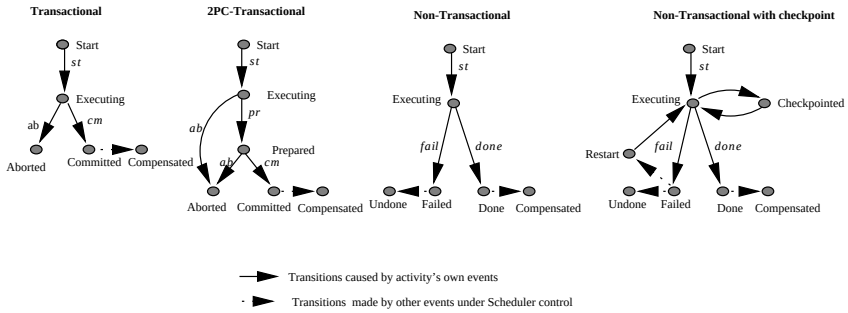

Figure 1: Typical task structures

```
IF (patient_id == 0)
  register(patient_id) COMPENSATED_BY delete(patient_id);
examine(patient_id, blood_test_type_list, roentgen_list);
AND_PARALLEL
{ blood(patient_id, blood_test_type_list, result1);
  WHILE (result2 == NULL)
    roent(patient_id, roentgen_list, result2); }
check(patient_id, result1, result2);
CONTINGENCY
{ cash(patient_id);
  credit(patient_id); } }
```

The above example is a simplified workflow of a check-up process carried out in a hospital. First, a patient is registered to the hospital, if she/he has not registered before. Then, she/he is examined by a doctor and according to the doctor's decision, a blood test is made and roentgen is taken for the patient in parallel. Since the patient need not wait for blood test to be finished in order roentgen to be taken, these two tasks are executed in an and_parallel block. Roentgen can be taken more than once, if the result is not clear. This is accomplished by an iterative block. After the results are checked by the doctor, the patient pays the receipt either in cash or by credit. These two tasks are placed in a contingency block so that, if the patient can not pay in cash, she/he is given the chance to pay by credit.

In METUFlow, there are five types of tasks. These are TRANSACTIONAL, NON_TRANSACTIONAL, NON_TRANSACTIONAL with CHECKPOINT, USER and 2PC_TRANSACTIONAL tasks. USER tasks are in fact NON_TRANSACTIONAL tasks. They are specified separately in order them to be handled by the worklist manager of the system. The states and transitions between the states for each of the task types are demonstrated in Figure 1. It should be noted that a task structure does not determine the means of execution nor the functionality of the task, but only a high level description of the visible state transitions.

We have chosen to include a second type of non_transactional task, namely, NON_TRANSACTIONAL with CHECKPOINT, in our model by making the observation that certain non_transactional tasks in real life, take checkpoints so that when a failure occurs, an application program rolls the task back to the last successful checkpoint.

These task types may have some attributes such as CRITICAL, NON_VITAL and CRITICAL_NON-_VITAL. Critical tasks can not be compensated and the failure of a non_vital task is ignored (Dayal and et.al. 1991, Chen and Dayal 1996). Besides these attributes, tasks can also have some properties like retriable, compensable, and undoable. A retriable task restarts execution depending on some condition when it fails. Compensation is used in undoing the visible effects of tasks after they are committed. Effects of an undoable task can be removed depending on some condition in case of failures. Some of these properties are special to specific task types. Undo conditions and tasks are only defined for non_transactional tasks, because transactional tasks do not leave any effects when they abort. Only 2PC_transactional tasks can be defined as critical. Note that the effects of critical tasks are visible to the other tasks in the workflow but the commitment of these tasks are delayed till the successful termination of the workflow. A task can be both critical and non_vital at the same time, but can not be critical and compensable. The example workflow definition of a check-up process illustrates the use of these different types of tasks.

In MFDL, activities in a process are declared using ACTIVITY reserved word. This declaration allows us to use activities sharing the same activity definition with different attributes and properties in the same workflow process.

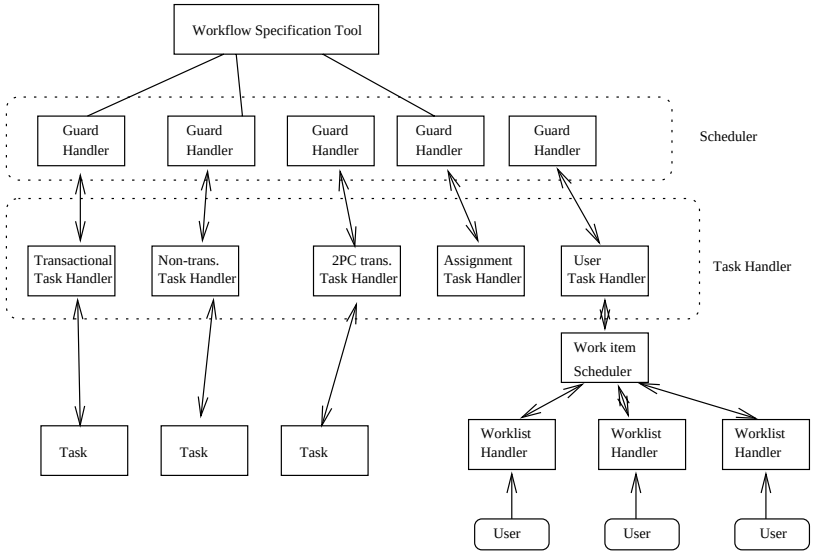

Figure 2: The simplified architecture of METUFlow

3 Distributed Scheduler of METUFlow

Workflow scheduler is the core component of a workflow management system. It instantiates processes according to the process description and controls correct execution of activities interacting with users via worklists and invoking applications as necessary.

A simplified architecture of METUFlow system is given in Figure 2. In METUFlow, first a workflow is specified using a graphical workflow specification tool which generates the textual workflow definition in MFDL as explained in Section 2. A process tree is then generated from this textual definition to determine the control-flow in terms of guards on events like start, abort and commit. Guards are expressions defined on events and occurrences of events are permitted only if their guards are true. Since METUFlow execution environment is distributed on the basis of activities, each activity should know when to start, abort or commit without consulting to a top-level central decision mechanism. For this purpose, a guard handler is associated with each activity instance which contains the guard expressions for the events of that activity instance.

During the construction of guards, the process tree of the workflow which consists of nodes representing processes, blocks and tasks is used. This tree explicitly shows the dependencies between the activities of the workflow. In Figure 3, the process tree corresponding to MFDL example of Section 2 is given. Each of the nodes is given a unique label to refer it in the execution phase. These activity labels make it possible for each task instance to have its own uniquely identified event symbols. The nodes shown in dashed lines are the compensation activities for the corresponding nodes. The guards of events are generated from the process tree in a straight forward way (Gokkoca et.al. 1997). As an example, guard expressions for start, commit and abort events of and_parallel block (labeled 5 in Figure 3) are given in the following:

Start: examine.commit

It means that and_parallel block can start only after *examine* commits.

Abort: blood.abort OR iterative.abort

When one of *blood* task or *iterative* block aborts, *and_parallel* block aborts.

Commit: blood.commit AND iterative.commit

and_parallel block can only commit if both *blood* task and *iterative* block commit.

Guard handlers generated for each of the activities evaluate the guard expressions like those given above through the event occurrence messages they receive. For example, when *examine* task commits, its guard handler notifies the guard handler of *and_parallel* block about the occurrence of this event and the start guard of *and_parallel* block evaluates to true.

There exists a task handler for each task instance which embodies a coarse description of the task instance including only the states and transitions (i.e. events) that are significant for coordination. The task handler acts as an interface between the task instance and its guard handler. A guard handler

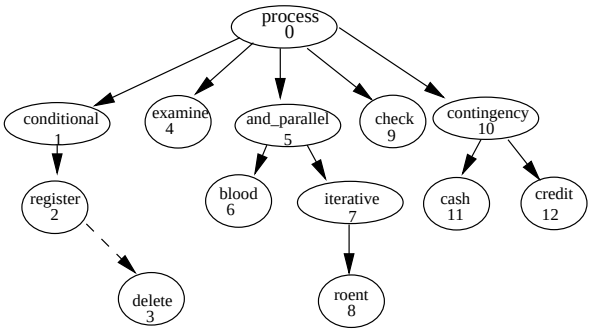

Figure 3: Process tree of the example MFDL

provides the message flow between the task's task handler and the other guard handlers in the system. According to the message it receives from the guard handler, a task handler causes the events related with that task to occur.

Each node in the process tree is implemented as a CORBA object with an interface for the guard handler to receive and send messages. At compile time the guards are generated and stored locally with the related objects. The objects to which the messages from this object are to be communicated are also recorded. A guard handler maintains the current guard for the events of the activity and manages communications. An event can happen only when its guard evaluates to true. If the guard for the attempted event is true, it is allowed right away. If it is false, it is rejected. Otherwise, it is parked. Parking an event means disabling its occurrence until its guard evaluates to true or false. When an event happens, messages announcing its occurrence are sent to the guard handlers of other related activities. Persistent queues are used to provide reliable message passing. When an event announcement arrives, the receiving guard handler simplifies its guard to incorporate this information. If the guard becomes true, then the appropriate parked event is enabled.

4 Task Handling in METUFlow

A task handler is created for each task instance. It acts as a bridge between the task and its guard handler. The guard handler sends the information necessary for the execution of the task, like the name of the task, parameters to the task handler and the task handler sends the information about the status of the task to the guard handler. When a task starts, its status becomes *Executing*. If it can terminate successfully, then its status is changed to *Committed* or *Done* depending on whether it is a transactional or a non-transactional task. In case the task fails, its status becomes *Abort* or *Failed*.

Task handler is a CORBA object and has a generic interface which contains the following methods to communicate with its associated guard handler:

- **Init** This method is used for passing initial data such as name of the task and initial parameters to the task handler.
- **Start** This method is called by the guard handler when the start guard of the task evaluates to true. This causes the task handler to invoke the actual task.

The task handlers for each different type of tasks inherit from this interface and provide overloading of these methods and/or further methods as necessary as explained in the following:

- **Transactional task handler** This type of task handler is coded for the transactional tasks. Even if a transactional task terminates successfully, its task handler should wait for the commit or abort message from the guard handler. For this purpose, in addition to the common methods described above, this type of task handler provides two more methods namely, *Commit* and *Abort* to be called by the guard handler when a task can commit or abort respectively.
- **Non-transactional task handler** This type of task handler handles tasks which are either non-transactional or non-transactional with checkpoint. The difference between non-transactional and non-transactional with checkpoint is that in the latter in case of a failure the application is rolled

```
register_patient()
{
  TaskExecuting();
   Connect_to_Database();

   /* This part of the code gets patient information from the user.
      A new patient_id is generated for this patient */

  Insert_Into_Database(patient_info,status);
  if(status == True){
     ReadyToCommit();
     if( GetStatus() == Commit) {
      Commit();
      TaskCommitted();
      Return(patient_id);
     } else {
      Abort();
      TaskAborted();
     }
  } else {
     Abort();
     TaskAborted();
  }
}
```

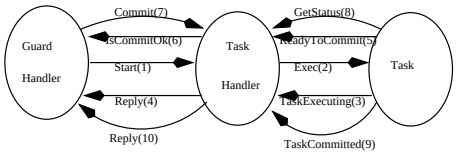

Figure 4: Communication among Guard Handler, Task Handler and Task

back to the latest checkpoint and not to the beginning. Since this does not affect the communication between the task and task handler, only one type of task handler is defined for both of them. Note that, non-transactional tasks terminate without waiting for any confirmation from the guard handler. They only inform the task handler about the status (*Done* or *Failed*).

- **Two phase commit task handler** This type of task handler is required for two phase commit transactional tasks. The difference between this type of task and transactional tasks is that, the former provides an additional status message, namely *Prepared*. Thus, this type of task handler provides a method called *Prepare* to be called by the transaction manager.

- **User task handler** This type of task handler is coded for the user tasks. User tasks are handled by work item scheduler and worklist handler (see Figure 2). The user task handler just stores the name of the task and the other necessary information to the repository from where the work item scheduler retrieves. The work item scheduler together with the worklist handlers informs the user about the tasks that she/he is responsible for and sends the status of the task to the user task handler.

- **Assignment task handler** This task handler does not cause any task to begin, but only a workflow relevant data assignment is done within the scope of a transaction.

In Workflow Reference Model of the Workflow Management Coalition (Hollingsworth 1994), the task handlers are classified according to having local or remote access. This classification is due to the assumption that the scheduler is centralized. Since scheduling is handled in a distributed manner in METUFlow, there is no need for such a classification.

The tasks may define their status in a way that the task handler can not understand or the task may not understand the messages coming from the task handler. Therefore, it becomes essential to interfere the source code of existing tasks. If it is possible to make changes in the task, then additional calls are added to the code of the task to convert the status information and error messages so that task handler and task can understand each other. If this is not possible, then the existing task is encapsulated by a code which provides the required conversion.

In Figure 4, we provide the modified code of the transactional task, register_patient, taken from the example given in Section 2 to illustrate the first strategy. The calls which are written in boldface are added to the original code of the task. The meanings of these calls are as follows:

- **TaskExecuting()** informs the task handler that it has started executing.

- **ReadyToCommit()** informs the task handler that operation is terminated successfully.

- **TaskAborted()** informs the task handler that the final status is Abort.
- **TaskCommited()** informs the task handler that the final status is Commit.
- **GetStatus()** checks the status message coming from the task handler.

In the figure, the labels of the arrows show the message passing between the entities. The labels are numbered according to the order in which the calls are made and the figure describes the flow of messages for the scenario in which the task terminates successfully and its commit guard evaluates to true. When the start guard of the task evaluates to true, the guard handler of the task calls **Start** method of the task handler. This causes the task handler to start execution of the task. When the task starts executing, as the first operation, task handler is informed by calling **TaskExecuting** call. The status of the task is sent to the guard handler by the task handler in **Reply** method with the parameter *Executing*. Then the normal flow of the task begins. If patient information is written to the database successfully, the task handler is sent **ReadyToCommit** call. The task handler informs the guard handler that the task is ready to commit by calling its **IsCommitOk** method. If the commit guard of the task evaluates to true, the guard handler informs task handler about this situation by calling its **Commit** method. Otherwise, **Abort** method is called. Task checks whether the message sent is *Abort* or *Commit* by the **GetStatus** call. If task handler sends *Commit*, then task commits actually and claims the final state as *Commit*. In case of *Abort* message, task aborts and sends the final abort status. When final status is claimed by the task, the task handler informs the guard handler about the final status by calling **Reply** method again.

5 Conclusion

In this paper, a block structured specification language, namely MFDL, and its scheduler implemented in a distributed environment through a CORBA compliant ORB are presented. The block structured procedural nature of MFDL provides both efficiency in execution and ease in testing and debugging. The task handling in the system is also discussed.

The future work includes incorporating correctness measures to guards, and providing efficient monitoring of the system.

References

Chen, Q., and Dayal, U., *"A Transactional Nested Process Management System"*, in Proceedings of the 12th International Conference on Data Engineering (ICDE'96), New Orleans, February 1996.

Dayal, U., Hsu M., and Ladin R., *"A Transactional Model for Long-running Activities"*, in Proceedings of the 17th International Conference on Very Large Data Bases (VLDB'91), Barcelona 1991.

Gokkoca, E., Altinel, M., Cingil, I., Tatbul, N., Koksal, P., and Dogac, A. *"Design and Implementation of a Distributed Workflow Enactment Service"*, In Proc. of Intl. Conf. on Cooperative Information Systems (CoopIS '97), Charleston, USA, June 1997.

Hollingsworth, D., *"The Workflow Reference Model"*, Workflow Management Coalition Specification, TC00-1003 (Draft 1.0), 1994.

Krishnakumar, N., and Sheth, A., *"Managing Heterogeneous Multi-System Tasks to Support Enterprise-Wide Operations"*, Distributed and Parallel Databases, 3(2):155-186, April 1995.

Object Management Group, *"The Common Object Request Broker: Architecture and Specification"*, OMG Document Number 91.12.1, December 1991.

Sheth, A., Georgakopoulos, D., Joosten, S., Rusinkiewicz, M., Scacchi, W., Wileden, J., Wolf, A., *"Report from the NSF Workshop and Process Automation in Information Systems"*, http://lsdis.cs.uga.edu/-activities/NSF-workflow, 1997.

Soley, R. M., and Stone, C. M., *"Object Management Architecture Guide"*, Third Edition, John Wiley & Sons, 1995.