Design and Implementation of a Distributed Workflow Management System: METUFlow *

Asuman Dogac, Esin Gokkoca, Sena Arpinar, Pinar Koksal, Ibrahim Cingil, Budak Arpinar, Nesime Tatbul, Pinar Karagoz, Ugur Halici, Mehmet Altinel

Software Research and Development Center
Dept. of Computer Engineering
Middle East Technical University (METU)
06531 Ankara Turkiye
asuman@srdc.metu.edu.tr

Summary. Workflows are activities involving the coordinated execution of multiple tasks performed by different processing entities, mostly in distributed heterogeneous environments which are very common in enterprises of even moderate complexity. Centralized workflow systems fall short to meet the demands of such environments.

This paper describes the design and implementation of a distributed workflow management system, namely, METUFlow. The main contribution of this prototype is to provide a truly distributed execution environment, where the scheduler, the history manager and the worklist manager of the system are fully distributed giving rise to failure resiliency and increased performance.

1. Introduction

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 may represent a manual operation by a human or a computerizable task to be invoked. Computerizable tasks may vary from legacy applications to programs to control instrumentation. 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.

This paper describes the design and implementation of a Workflow Management System prototype, namely METUFlow. METUFlow handles the interoperability of applications on heterogeneous platform by using CORBA as the communication infrastructure. The distinguishing features of METUFlow are: a distributed scheduling mechanism with a distributed history and a distributed worklist management. 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

* This work is partially being supported by the Middle East Technical University, Project Number: AFP-97-07-02-08, 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 and by Sevgi Holding (Turkey).

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. In order to fully exploit the advantages brought by the distributed scheduling, METUFlow history management, workflow relevant data management and the worklist management are also handled in a distributed manner.

Data inconsistency can be violated by improper interleaving of concurrently executing workflows. Therefore, there should exist a mechanism to prevent such interleavings to ensure data consistency in workflow management systems (WFMS). In METUFLow, we introduce the concept of "sphere of isolation" for the correctness of concurrently executing workflows while increasing concurrency. Spheres of isolations are obtained by exploiting the available semantics in workflow specification.

METUFlow has a block structured specification language, namely METUFlow Definition Language (MFDL). The advantages brought by this language can be summarized as follows:

- As noted in [**?**], 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. MFDL prevents these disadvantages.
- A block structured language confines the intertask dependencies to a well formed structure which in turn proves extremely helpful in generating the guards of events for distributed scheduling of the workflow.
- A block not only clearly defines 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.

The paper is organized as follows: Section 2 describes the METUFlow Specification Language. In Section 3, we give the underlying mechanism of METUFlow scheduler. The architecture of METUFlow is provided in Section 4. In the sections that follow namely Section 5, 6, 7, 8 and 9, the details of the components of METUFlow, namely, scheduler, task handlers, worklist manager, history manager and transaction manager are explained. In Section 10, our concurrency control mechanism is described. Finally, the conclusions are provided in Section 11.

2. The Process Model and the METUFlow Definition Language: MFDL

In a workflow definition language, the tasks involved in a business process, the execution and data dependencies between these tasks are provided.

METUFlow Definition Language (MFDL) that we have designed has a graphical user interface developed through Java which allows defining a workflow by accessing METUFlow from any computer that has a Web browser. This feature of METUFlow makes it possible to support mobile users.

The WfMC have identified a set of six primitives with which to describe flows and hence construct a workflow specification [**?**]. With these primitives it is possible to model any workflow that is likely to occur. These primitives are: sequential, AND-split, AND-join, OR-split, OR-join and repeatable task. These primitives are all supported by MFDL through its block types. MFDL contains seven types of blocks, namely, serial, and_parallel, or_parallel, xor_parallel, contingency, conditional and iterative blocks. Of the above block types, serial block implements the sequential primitive. And_parallel block models the AND-split and AND-join primitives. AND-split, OR-join pair is modelled by or_parallel block. Conditional block corresponds to OR-split and OR-join primitives. Finally, Repeatable task primitive is supported by the iterative block.

A workflow process is defined as a collection of blocks, tasks and 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. The term *activity* is used to refer to a block, a task or a (sub)process. Blocks differ from tasks and processes in that they are conceptual activities which are used only to specify the ordering and the dependencies between activities.

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

Syntax 1 B = $(A_1; A_2; A_3; .....; A_n)$, where B is a serial block.

Definition 1 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.

Syntax 2 B = $(A_1$ & A_2 & ..... & $A_n)$, where B is an and_parallel block.

Definition 2 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.

Syntax 3 B = $(A_1|A_2|.....|A_n)$, where B is an or_parallel block.

Definition 3 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.

Syntax 4 B = $(A_1||A_2||.....||A_n)$, where B is an xor_parallel block.

Definition 4 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 the other activities in the block to abort. If all of the activities abort, the block aborts.

Syntax 5 B = $(A_1, A_2, ...A_n)$, where B is a contingency block.

Definition 5 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.

Syntax 6 B = (condition,A_1, A_2), where B is a conditional block.

Definition 6 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.

Syntax 7 B = (condition;$A_1; A_2; .....; A_n$), where B is an iterative block.

Definition 7 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 is true, then A_1 starts executing again. If one of the activities aborts at any one of the iterations, B aborts.

Syntax 8 A = (A_c, AbortList(A_c)), where A_c is the compensation activity of A.

Definition 8 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 during compilation 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.

Syntax 9 T = T_u, where T_u is the undo task of task T.

Definition 9 The undo task T_u of T starts if T fails.

In addition to activities, there is an assignment statement in MFDL which accesses and updates the workflow relevant data.

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[20],
      OUT int roentgen_list[20])
```

```
      PARTICIPANT DOCTOR;

USER_ACTIVITY  blood_exam (IN int patient_id,
      IN int  blood_test_type_list[20], OUT STRING result[20])
      PARTICIPANT LABORANT;
USER_ACTIVITY  roentgen (IN int patient_id,
      IN int roentgen_list[20], OUT STRING result[20])
      PARTICIPANT ROENTGENOLOGIST;
USER_ACTIVITY  check_result (IN int patient_id,
      IN  string result1[20], IN STRING result2[20])
      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 int patient_id;
 var STRING result1[20], result2[20];
 var int  blood_test_type_list[20], roentgen_list[20];

 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);
 XOR_PARALLEL
 {
      cash(patient_id);
      credit(patient_id);
 }
}
```

This 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

Table 2.1. Event Attributes

activity_types	start	abort/fail	commit/done
transactional	triggerable	immediate	normal
2PC_transactional	triggerable	normal,immediate	normal
non_transactional	triggerable	immediate	immediate
non_transactional with checkpoint	triggerable	immediate	immediate

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 an xor_parallel block so that, cash and credit begins in parallel and commit of one causes the other to abort.

In METUFlow, there are five types of tasks. These are TRANSACTIONAL, NON_TRANSACTIONAL, NON_TRANSACTIONAL with CHECKPOINT, USER and 2PC_TRANSACTIONAL activities. USER activities are in fact NON_TRANSACTIONAL activities. They are specified separately in order to be used by the worklist manager which handles the user-involved activities. The states and transitions between these states for each of the activity types are demonstrated in Figure 2.1. The significant events in METUFlow are start, commit and abort. The event attributes of these tasks are shown in Table 2.1. They are taken into account during guard generation. Normal events are delayable and rejectable (e.g. commit), inevitable events are delayable and nonrejectable, immediate events are nondelayable and nonrejectable (e.g. abort), and triggerable events are forcible (e.g. start).

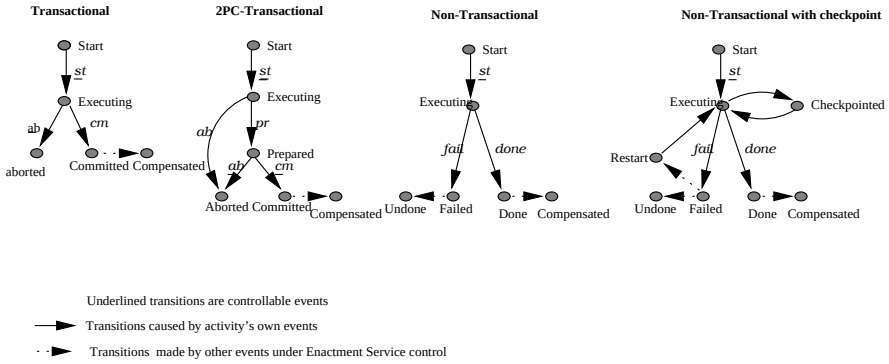


Fig. 2.1. Typical task structures

Note that the abort event of a 2PC_TRANSACTIONAL task after the coordinator has taken a decision is normal whereas it is immediate before the coordinator has taken a decision. Triggerable and normal events are controllable because they can be triggerred, rejected or delayed while immediate events are uncontrollable. We have chosen to include a second type of non_transactional activity, namely, NON_TRANSACTIONAL with CHECKPOINT, in our model by making the observation that certain non_transactional activities in real life, take checkpoints so that when a failure occurs, an application program rolls the activity back to the last successful checkpoint.

These activity types may have some attributes such as CRITICAL, NON_VITAL and CRITICAL_NON_VITAL. Critical activities can not be compensated and the failure of a non_vital activity is ignored [?, ?]. Besides these attributes, activities can also have some properties like retriable, compensable, and undoable. A retriable activity restarts execution depending on some condition when it fails. Compensation is used in undoing the visible effects of activities after they are committed. Effects of an undoable activity can be removed depending on some condition in case of failures. Some of these properties are special to specific activity types. Undo conditions are only defined for non_transactional tasks, because transactional tasks do not leave any effects when they abort. Only 2PC_transactional activities can be defined as critical. Note that the effects of critical activities are visible to the other activities in the workflow but the commitment of these activities are delayed till the successful termination of the workflow. An activity can be both critical and non_vital at the same time, but can not be critical and compensable.

In MFDL, activities in a process are declared using the reserved word ACTIVITY. This declaration allows the sharing of an activity definition among many workflow processes with possibly different attributes and properties for each instance.

3. Guard Generation for Distributed Scheduling

In this section, first the semantics of the block types are defined using ACTA formalism. We then show that the two dependencies provided in [?] are adequate to express the specified block semantics and result in simple guard expressions. Finally, a mechanism for guard construction is presented.

3.1 Semantics of the Block Types using ACTA Formalism

We use the ACTA formalism [?, ?] with slight modifications to express the semantics of block types[1] as follows:

Let t_i and t_j be two transactions.

[1] We treat fail/done event of non_transactional activities as abort/commit of transactional activities.

- **Commit Dependency**(t_j CD t_i): if transaction t_i commits, then t_j commits.
- **Commit-on-Abort Dependency**(t_j CAD t_i): if transaction t_i aborts, then t_j commits.
- **Abort Dependency**(t_j AD t_i): if transaction t_i aborts, then t_j aborts.
- **Abort-on-Commit Dependency**(t_j ACD t_i): if transaction t_i commits, then t_j aborts.
- **Begin Dependency**(t_j BD t_i): if transaction t_i begins executing, then t_j starts.
- **Begin-on-Commit Dependency**(t_j BCD t_i): if transaction t_i commits, then t_j begins executing.
- **Begin-on-Abort Dependency**(t_j BAD t_i): if transaction t_i aborts, then t_j begins executing.

Conditional dependencies are added to the ACTA formalism. These dependencies have an additional argument which is "condition". For example, a conditional begin dependency is expressed as BD(C). If condition C is true, then BD holds, else it does not hold.

Using the modified ACTA formalism, semantics of block types can be restated as follows:

Semantics 1 B = $(A_1; A_2; A_3; .....; A_n)$, where B is a serial block.
- A_1 BD B
- A_{i+1} BCD A_i , 1 ≤i<n
- B CD A_n
- B AD A_i ,1 ≤ i ≤n

Semantics 2 B = $(A_1$ & A_2 & ..... & $A_n)$, where B is an and_parallel block.
- A_i BD B ,1 ≤ i ≤n
- B AD A_i ,1 ≤ i ≤n
- ∀i(B CD A_i)

Semantics 3 B = $(A_1|A_2|.....|A_n)$, where B is an or_parallel block.
- A_i BD B ,1 ≤ i ≤n
- ∃i (B CD A_i) ∧ (∀j((B CD A_j) ∨ (B CAD A_j))), j≠i
- ∀i(B AD A_i)

Semantics 4 B = $(A_1||A_2||.....||A_n)$, where B is an xor_parallel block.
- A_i BD B ,1 ≤ i ≤n
- ∃i (B CD A_i) ∧ (∀j(A_j ACD A_i)), i≠j
- ∀i(B AD A_i)

Semantics 5 B = $(A_1, A_2, ...A_n)$, where B is a contingency block.
- A_1 BD B
- A_{i+1} BAD A_i, 1 ≤ i<n
- B CD A_i, 1 ≤i≤n
- B AD A_n

Semantics 6 B = (condition(C),A_1, A_2), where B is a conditional block.
- A_1 BD(C) B

- A_2 BD(¬C) B
- B CD(C) A_1
- B CD(¬C) A_2
- B AD(C) A_1
- B AD(¬C) A_2

Semantics 7 B = (condition(C);$A_1; A_2; .....; A_n$), where B is an iterative block.

- A_1 BD(C) B
- A_{i+1} BCD A_i, 1 ≤i<n
- B CD(¬C) A_n
- B AD A_i

Semantics 8 A = (A_c, AbortList(A_c)), where A_c is the compensation activity of A.

- (A_c BCD A) ∧ (A_c BAD AbortList(A_c))

Semantics 9 T = T_u, where T_u is the undo task of T.

- T_u BAD T

ACTA formalism specifies the transaction semantics of a model by presenting transaction relations with predefined dependencies. However, these dependencies are expressed at the abstract level and therefore the following two primitives [**?**, **?**] are used to specify intertask dependencies as constraints on the occurrence and temporal order of events:

1. $e_1 \rightarrow e_2$: If e_1 occurs, then e_2 must also occur. There is no implied ordering on the occurrence of e_1 and e_2.
2. $e_1 < e_2$: If e_1 and e_2 both occur, then e_1 must preceed e_2.

The ACTA dependencies used in the specification of the block semantics are expressed in terms of these two primitives as follows:

- **Commit Dependency**(t_j CD t_i):
  $(Commit_{t_j} \rightarrow Commit_{t_i}) \wedge (Commit_{t_i} < Commit_{t_j})$
- **Commit-on-Abort Dependency**(t_j CAD t_i):
  $(Abort_{t_j} \rightarrow Commit_{t_i}) \wedge (Commit_{t_i} < Abort_{t_j})$
- **Abort Dependency**(t_j AD t_i):
  $(Abort_{t_j} \rightarrow Abort_{t_i}) \wedge (Abort_{t_i} < Abort_{t_j})$
- **Abort-on-Commit Dependency**(t_j ACD t_i):
  $(Abort_{t_j} \rightarrow Commit_{t_i}) \wedge (Commit_{t_i} < Abort_{t_j})$
- **Begin Dependency**(t_j BD t_i):
  $(Start_{t_j} \rightarrow Start_{t_i}) \wedge (Start_{t_i} < Start_{t_j})$
- **Begin-on-Commit Dependency**(t_j BCD t_i):
  $(Start_{t_j} \rightarrow Commit_{t_i}) \wedge (Commit_{t_i} < Start_{t_j})$
- **Begin-on-Abort Dependency**(t_j BAD t_i):
  $(Start_{t_j} \rightarrow Abort_{t_i}) \wedge (Abort_{t_i} < Start_{t_j})$

The guards of events corresponding to these two primitive dependencies are as follows [**?**, **?**]:

Table 3.1. Guards corresponding to the dependency set

dependency	e	f	G(f)	G(e)
A BD B	B_{st}	A_{st}	$\Box B_{st}$	TRUE
A BCD B	B_{cm}	A_{st}	$\Box B_{cm}$	TRUE
A BAD B	B_{ab}	A_{st}	$\Box B_{ab}$	TRUE
A CD B	B_{cm}	A_{cm}	$\Box B_{cm}$	TRUE
A CAD B	B_{ab}	A_{cm}	$\Box B_{ab}$	TRUE
A AD B	B_{ab}	A_{ab}	$\Box B_{ab}$	TRUE
A ACD B	B_{cm}	A_{ab}	$\Box B_{cm}$	TRUE

For the constraint $e < f$, which corresponds to the dependency $D_< = \bar{e} \vee \bar{f} \vee e \bullet f$, the guards are:

- $\mathcal{G}(e) = \mathsf{TRUE}$
- $\mathcal{G}(f) = \Diamond \bar{e} \vee \Box e$

Note that $\Box e$ means that e will always hold; $\Diamond e$ means that e will eventually hold (thus $\Box e$ entails $\Diamond e$). At runtime e can occur at any point in the history whereas f can occur only if e has occurred or it is guaranteed that $\bar{e}$ will occur.

For the constraint $f \rightarrow e$, which corresponds to the dependency $D_\rightarrow = \bar{f} \vee e$, the guards of events are:

- $\mathcal{G}(e) = \mathsf{TRUE}$
- $\mathcal{G}(f) = \Diamond e$

These guards state that e can occur at any time in the history; f can occur if e has happened or will happen.

3.2 Guard Construction Steps

We use the dependencies BD, BCD, BAD to compute start guards, AD, ACD to generate abort guards and CD, CAD to compute commit guards of activities [?]. Note that all of these dependencies are in the form of an expression which contains one subexpression with $\rightarrow$ primitive and the other with $<$ primitive with a conjunction in between them such as $(f \rightarrow e) \wedge (e < f)$. We present the construction of guards of events e and f for this dependency in the following [?]:

$\mathcal{G}(e) = \text{TRUE}$

$\mathcal{G}(f) = \mathcal{G}(D_\rightarrow, f) \wedge \mathcal{G}(D_<, f) = \Diamond e \wedge (\Diamond \bar{e} \vee \Box e) = (\Diamond e \wedge \Diamond \bar{e}) \vee (\Diamond e \wedge \Box e) = FALSE \vee (\Diamond e \wedge \Box e) = \Diamond e \wedge \Box e = \Box e$

Note that after simplification, the guard of f turned out to be $\Box e$. In other words, the occurrence of event f only requires event e to have already happened. This is an intuitively expected result since in our workflow specification, the occurrence of events only depends on the events already occurred

with no references to the future events. This result facilitates the computation of the guards drastically. The guards of events of the dependency set corresponding to our workflow specification language are computed as presented in Table 3.1. Note that from this result, we conclude that if we want to compute the guard related to an activity A_1, we must consider only "A_1 ACTA_Dep A_2" type dependencies, not "A_2 ACTA_Dep A_1" type dependencies. The reason is that in the latter, the guard of any event related with A_1 is already TRUE from Table 3.1.

If we summarize, by starting with a block structured workflow specification language, we obtain a well defined set of dependencies, all in the form $(f \rightarrow e) \wedge (e < f)$. This produces very straightforward guards for events which in turn, makes it possible to compute the guards directly from the process definition with a simple algorithm. The complete guard generation process is outlined in Figure 3.1.

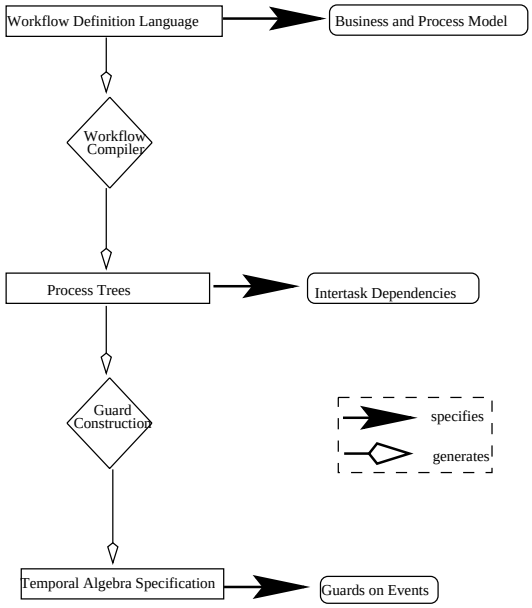


Fig. 3.1. Guard generation process

A process tree is generated from the workflow specification in MFDL. The process tree consists of nodes representing processes, blocks and tasks, and is used only during compilation time, execution being completely distributed. 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. This tree explicitly shows the dependencies between the activities of the workflow. In fact, with Table 3.1 at hand, it is possible to generate the guards of a process from its process tree. In the following we describe the guard construction process through an example.

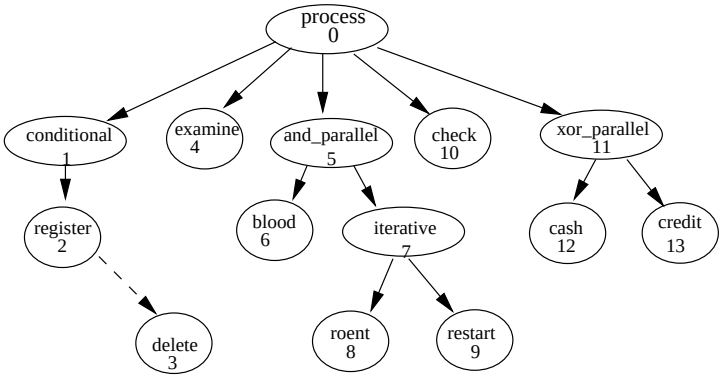


Fig. 3.2. Process tree of the example MFDL

In Figure 3.2, the process tree corresponding to MFDL example of Section 2 is given. The nodes shown in dashed lines are the compensation activities for the corresponding nodes.

Consider node 3 of Figure 3.2 which is a compensation task. Using Semantics 8:

D_1 : 3 BAD 0
D_2 : 3 BCD 2
D : $D_1 \wedge D_2$

Note that AbortList of 3 is {0}, because the compensation of 2 is needed only when 0 aborts. From Table 3.1,

G(D_1, 3_{st}) = $\Box 0_{ab}$
G(D_2, 3_{st}) = $\Box 2_{cm}$
G(D, 3_{st}) = G(D_1, 3_{st}) $\wedge$ G(D_2, 3_{st}) = $\Box 0_{ab} \wedge \Box 2_{cm}$.

This guard states that task 3 should be started when process itself (node 0) is aborted while task 2 has committed.

In Figure 3.2, there is a restart node labeled as 9. This node is special to iterative block. Restart node is treated like the other children of the iterative node during execution. Its role is to prepare the block for the next iteration while the iteration condition is true. After restart node commits, the iteration condition is checked. If it is true, the next iteration starts. Otherwise, the iterative node commits, as stated in Semantics 7 (A_n corresponds to restart node). Note that this cyclic dependency in arbitrary tasks is handled in [?] by resurrecting a guard under appropriate conditions. Ours is a practical implementation of this formal concept.

Table 3.2 shows the start, abort and commit guards for all the nodes of the example process tree given in Figure 3.2.

Table 3.2. Guards of the example workflow definition

label	start	start condition	abort	commit	commit condition
0	TRUE		$\Box 1_{ab} \vee \Box 4_{ab} \vee \Box 5_{ab} \vee \Box 10_{ab} \vee \Box 11_{ab}$	$\Box 11_{cm}$	
1	$\Box 0_{st}$	patient_id == 0	$\Box 2_{ab}$	$\Box 2_{cm}$	
2	$\Box 1_{st}$		TRUE	TRUE	
3	$\Box 0_{ab} \wedge \Box 2_{cm}$		TRUE	TRUE	
4	$\Box 1_{cm}$		TRUE	TRUE	
5	$\Box 4_{cm}$		$\Box 6_{ab} \vee \Box 7_{ab}$	$\Box 6_{cm} \wedge \Box 7_{cm}$	
6	$\Box 5_{st}$		TRUE	TRUE	
7	$\Box 5_{st}$	result2 == Null	$\Box 8_{ab} \vee \Box 9_{ab}$	$\Box 9_{cm}$	result2 != Null
8	$\Box 7_{st}$		TRUE	TRUE	
9	$\Box 8_{cm}$		TRUE	TRUE	
10	$\Box 5_{cm}$		TRUE	TRUE	
11	$\Box 10_{cm}$		$\Box 12_{ab} \wedge \Box 13_{ab}$	$\Box 12_{cm} \vee \Box 13_{cm}$	
12	$\Box 11_{st}$		$\Box 13_{cm}$	$\Box 13_{ab}$	
13	$\Box 11_{st}$		$\Box 12_{cm}$	$\Box 12_{ab}$	

It should be noted that in Table 3.2, some of the guards are set to TRUE right away. This is because either the occurrences of these events do not depend on the occurrence of any event or they are immediate events. Also note that, xor_parallel blocks identify a race condition without a need for preprocessing. For example, from Table 3.2, it is clear that abort of 12 is dependent on the commitment of 13 and commitment of 13 is dependent on the abort of 12. Obviously, this creates a deadlock situation. We implemented a modified 2 Phase Commitment protocol to handle this case (See Section 9.). When xor_parallel block starts, all of its immediate children are registered to the coordinator object belonging to this block. The coordinator keeps track of status of these children to ensure that only one of them commits. In this case, the abort and commit guards are not constructed any more for the child nodes.

4. METUFlow Architecture

A simplified architecture of METUFlow system is given in Figure 4.1. 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. The core component of a workflow management system is the workflow scheduler which instantiates workflows according to the workflow specification and controls correct execution of activities interacting

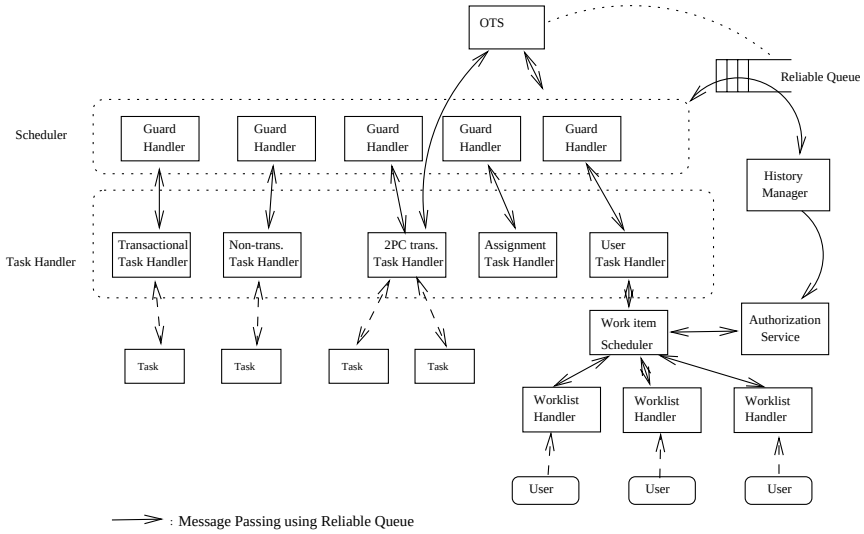


Fig. 4.1. The simplified architecture of METUFlow

with users via worklists and invoking applications as necessary. In METUFlow, the functionality of the scheduler is distributed to a number of guard handlers which contain the guard expressions for the events of the activity instances as explained in Section 5. Also, there exist a task handler which acts as an interface between the activity instance and its guard handler. Details of task handling in METUFlow is discussed in Section 6. In a workflow management system, there are activities in which human interactions are necessary. In METUFlow, work item scheduler manages such interactions. It is responsible for progressing work requiring user attention and interacts with the scheduler through user task handler as shown in Figure 4.1. Work item scheduler uses the authorization service to determine the authorized roles and users. The detailed architecture of work item scheduler is provided in Section 7. History Manager provides the mechanisms for storing and querying the history of both ongoing and past processes. It communicates with the scheduler through a reliable message queue to keep track of the execution of processes. It is also necessary for history manager to be in touch with the authorization service to inform about the past processes that affect security.

The communication infrastructure of METUFlow is CORBA but CORBA does not provide for reliable message passing, that is, when ORB crashes, all of the transient messages are lost. For this reason, we have implemented a reliable message passing mechanism which uses Object Transaction Service (OTS) based transaction manager (See Section 9.) to commit distributed transactions. Note that reliable message passing is necessary among all the

components of METUFlow such as between guard handlers and task handlers as indicated in Figure 4.1.

5. Guard Handlers in METUFlow

After textual workflow definition is produced by MFDL, a process tree is generated using this textual definition as explained in Section 3.2. A guard expression is generated for each node of the process tree. After the guards are constructed, an environment in which these guards are evaluated through the event occurrence messages they receive is created. 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 [?]. Also, there exists a task handler for each activity instance which embodies a coarse description of the activity instance including only the states and transitions (i.e. events) that are significant for coordination. A guard handler provides the message flow between the activity's task handler and the other guard handlers in the system.

Each node in the process tree is implemented as a CORBA [?, ?] object with an interface for the guard handler to receive and send messages. Figure 5.1 shows the execution environment of objects of guard handler for the example check-up workflow. The reason for creating objects for each node rather than only for leaf nodes, which correspond to the actual tasks, is that carrying block semantics to the execution reduces the number of messages to be communicated. This is explained in the following example:

Assume that we have a process segment like:

```
serial {
       and_parallel {
            T1();
            T2();
            ...
            Tn();
       }
       and_parallel {
            T1();
            T2();
            ...
            Tn();
       }
}
```

Without a block abstraction during execution, the start guard of each activity in the second and_parallel block must contain the commit event of each task of the first and_parallel block. Obviously this necessitates to communicate the commit event of each of the n tasks in the first and_parallel

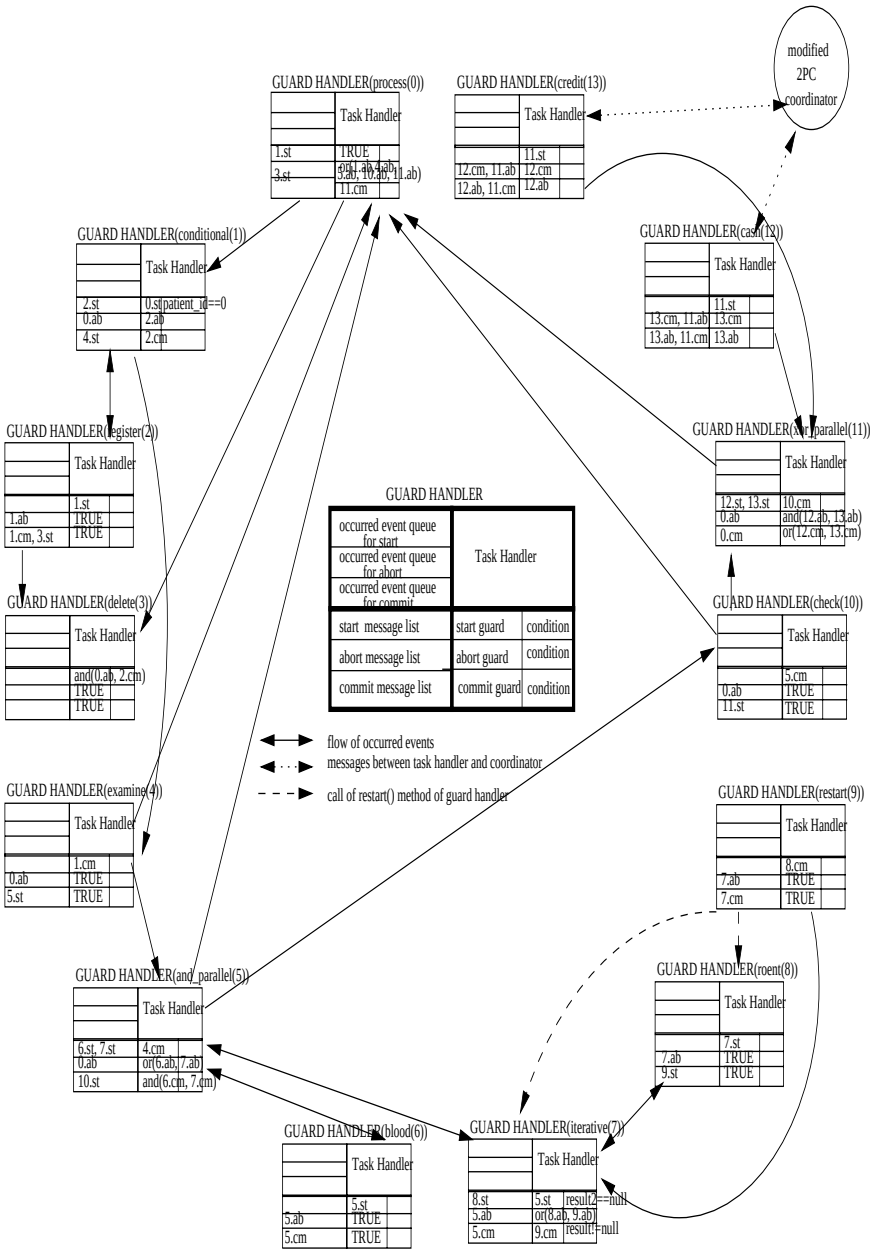


Fig. 5.1. Execution environment of objects of Guard Handler

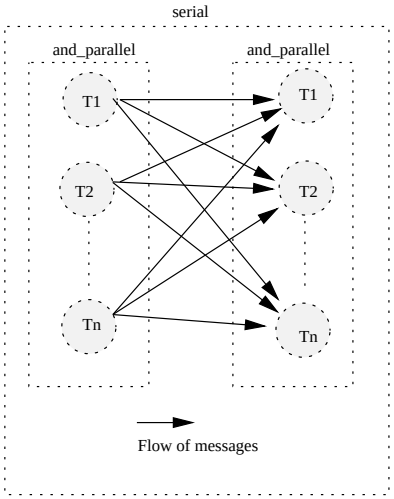


Fig. 5.2. Environment of objects without block abstraction

block to each of the n tasks in the second and_parallel block. Hence without a block abstraction, the number of messages to be communicated is n^2, as shown in Figure 5.2.

When block abstraction is used during execution as shown in Figure 5.3, the start guard of the second and_parallel block contains the commit event of the first and_parallel block. Thus the commit guard of the first and_parallel block contains the commit events of each of its n tasks; the start guards of each of the tasks in the second and_parallel block contain the start event of the second and_parallel block. For this case, the number of messages communicated reduces to 2n + 1, as shown in Figure 5.3.

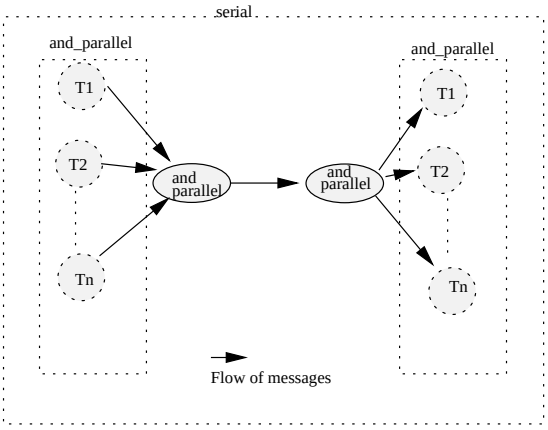


Fig. 5.3. Environment of objects with block abstraction

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. For example for task 3, since its start guard contains an abort event of the process, the abort_message_list of the process contains the object identifier of task 3 to indicate that the start guard of task 3 should be informed of the abort of the process. When an object receives an event to be consumed, it is placed in the occurred_events_queue of the related significant event of the object. Figure 5.1 explicitly shows the source and the destination of the messages.

A guard handler maintains the current guard for the significant events of the activity and manages communications. When a task handler is ready to make a transition, it attempts the corresponding event. Intuitively, 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 defering its occurrence until its guard simplifies 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.

6. 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 and changed values of the parameters 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 *Aborted* 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 task 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. Therefore, in addition to the common methods described above, this type of task handler provides two more methods, *Commit* and *Abort* to be called by the guard handler when a task is allowed to commit or abort respectively.
- **Non-transactional task handler.** This type of task handler handles tasks which are of type 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 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 4.1). 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 (See Section 7.).
- **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 [**?**], 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.

```
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();
  }
}
```

Fig. 6.1. An Example Task Code

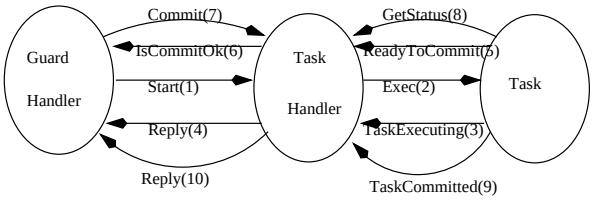


Fig. 6.2. Communication among Guard Handler, Task Handler and Task

In Figure 6.1, 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.

– **TaskCommitted()** informs the task handler that the final status is Commit.
– **GetStatus()** checks the status message coming from the task handler.

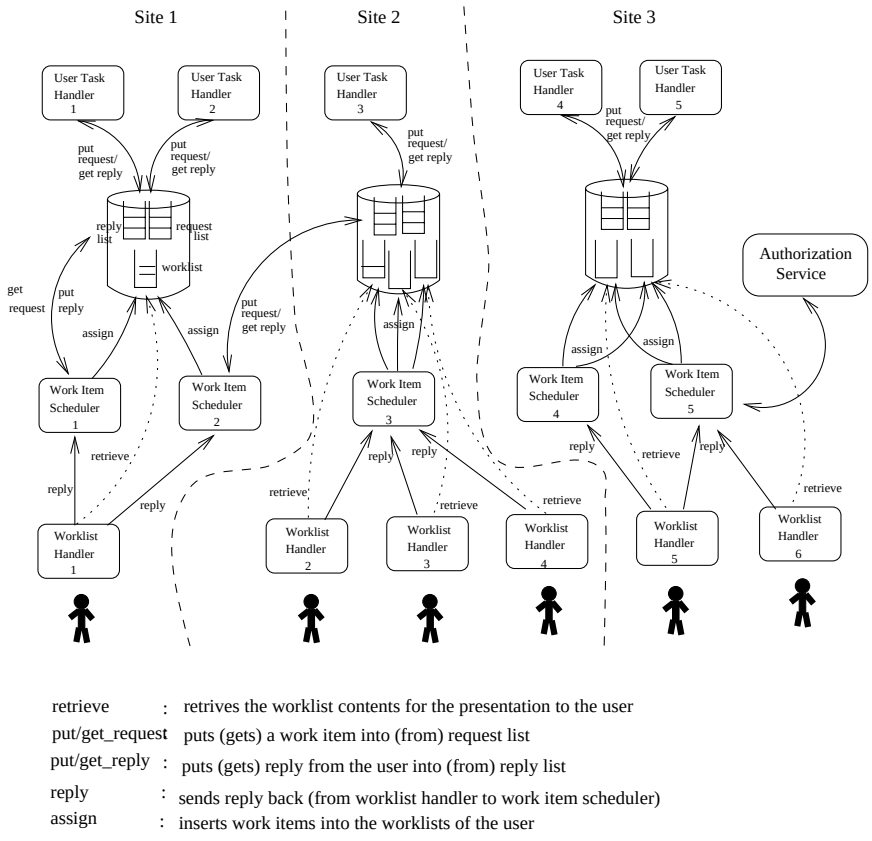
The communication mechanism among guard handler, task handler and a task is provided in Figure 6.2. In the figure, the labels of the arrows show the message passing between the entities. The labels are numbered according to the order in which the calls are made and the figure describes the flow of messages 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.

7. Worklist Management in METUFlow

The worklist manager is a software component which manages the interaction between workflow participants and the scheduler.

In METUFlow, the worklists are distributed, that is, a worklist at a site contains the work items to be accessed by the users at that site.

When a user activity is to be invoked by the scheduler, a user task handler created for this purpose stores the request (work item) into a request list within the scope of a transaction. Request list is a CORBA object and its implementation in a particular site depends on the persistent storage available in that site, that is, this CORBA object is implemented on a DBMS if it is available, otherwise it is implemented as a file. Worklist manager, as depicted in Figure 7.1 consists of two components. The first one, work item scheduler, decides on the assignment of work items to the worklists of the users in cooperation with the authorization service. The first version of the authorization service implemented contains the definitions of roles and their members, authorizations to execute tasks and constraints controlling



retrieve : retrives the worklist contents for the presentation to the user
put/get_request puts (gets) a work item into (from) request list
put/get_reply : puts (gets) reply from the user into (from) reply list
reply : sends reply back (from worklist handler to work item scheduler)
assign : inserts work items into the worklists of the user

Fig. 7.1. Worklist Manager in METUFlow

the execution of these authorizations. We plan to improve this service by involving periodic, temporal, event based distributed constrained authorizations and authentication services. The work item scheduler is also responsible for putting the reply back into the reply list, again, within the scope of a transaction. That is, the reply list is a persistent CORBA object whose implementation is realized through a DBMS or a file depending on the capabilities of the site concerned. The second component, worklist handler is responsible for retrieving work items to be presented to the user for processing.

A point to be noted over here is the following: CORBA provides location transparency, in other words the users need not be aware of the location of

the objects to be created. However, CORBA does provide mechanisms to affect the object creation site although the specifics depends on the ORB at hand. First by default, an object is created at the local site if it is possible. Therefore, whenever there is a request to create a work item scheduler, it is created at the same site with the user task handler. In order to be able to create worklists at the same host with the involved user (or role), a list is kept which stores the association between the user-ids and host-ids. In METUFlow lookup method of Orbix's locator class ([?]) is used for this purpose.

Finally in order to provide access to the worklists through World-Wide-Web we have chosen to implement them in Java which made it easier to connect to a CORBA compliant ORB, namely Orbix through OrbixWeb.

8. History Management in METUFlow

Workflow history management provides the mechanisms for storing and querying the history of both ongoing and past processes. This serves two purposes: First, during the execution of a workflow, the need may arise for looking up some piece of information in the process history, for example, to figure out who else has already been concerned with the workflow at what time in what role, and so on. This kind of information contributes to more transparency, flexibility and overall work quality. Second, aggregating and mining the histories of all workflows over a longer time period forms the basis for analyzing and assessing the efficiency, accuracy, and the timeliness of the enterprise's business processes. So, this information provides the feedback for continuous business processes re-engineering. Given that, much of the history information relates to the time dimension in that it refers to turnaround times, deadlines, delays, etc. over a long time horizon.

In consistence with its architecture, METUFlow history and workflow relevant data handling mechanism is based on CORBA. The history of each activity instance is implemented as a CORBA object. To exploit the advantages brought by the distributed execution of the workflow scheduler, history management should also be distributed. To make distributed history management possible, the persistent store in which the history information is kept, should also be distributed over the network.

The history of each activity instance is implemented as a CORBA object at the same site at which the activity object itself is invoked. If a DBMS is available at the concerned site, it is used as the persistent store, otherwise a binary file is used for this purpose. It is possible to have history objects created at the same site where they are activated to prevent the communication cost with the activity instance objects.

Each activity instance is responsible for its own history object and knows the object identifier of its parent activity instance. A child activity instance invokes a method to pass the object identifier of its own history object to its parent object. A parent activity instance object establishes the links between

its own history object and its child's history object. Note that in the eventual history tree of the process instance obtained this way objects are linked through their object identifiers according to the process tree.

In summary with a distributed history and workflow relevant data handling mechanism, availability and scalability aspects of the system are increased.

When it comes to querying history both for monitoring and for data mining purposes, having encapsulated these data as CORBA objects naturally yields to using the Query Service Specification of OMG.

The Query Service provides query operations on collection of objects. The Query Service can be used to return collections of objects that may be:

- selected from source collections based on whether their member objects satisfy a given predicate.
- produced by query evaluators based on the evaluation of a given predicate. These query evaluators may manage implicit collections of objects.

A Query Evaluator with temporal dimension is being developed for this purpose within the scope of the METUFlow project [**?**].

9. OTS Based Transaction Manager

In METUFlow, distributed transaction management is realized through a transaction manager that implements Object Transaction Service (OTS) Specification of OMG, OTS [**?**]. OTS specification describes a service that supports flat and nested transactions in a distributed heterogeneous environment. It defines interfaces that allow multiple, distributed objects to cooperate to provide atomicity of transactions. These interfaces enable the objects either commit or rollback all the changes together in the presence of failure.

Figure 9.1 illustrates the major components and the interfaces defined by OTS. In a typical scenario, a transactional client (transaction originator) creates a transaction obtaining a Control Object from a Factory provided by ORB. Transaction clients uses the Current pseudo-object to begin a transaction, which becomes associated with the transaction originator's thread. The Current interface defines operations that allow a client of OTS to begin and end transactions and to obtain information about the current transaction. A simplified version of Current interface is illustrated below:

```
interface Current {
 void begin();
 void commit();
 void rollback();
 Status get_status();
 string get_transaction_name();
 ...
}
```

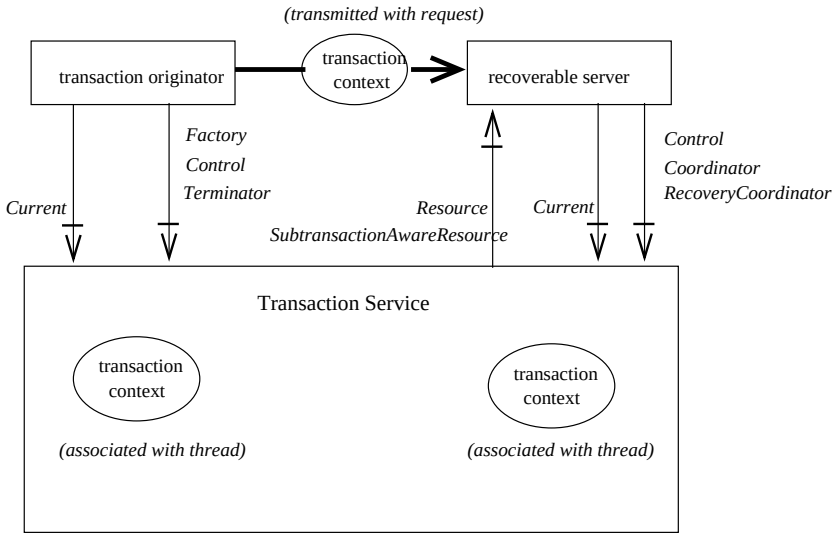


Fig. 9.1. The major components and interfaces of the OTS

ORB associates a Transaction Context with each Control object. A transaction context contains all the necessary information to control and to coordinate transactions. Transaction context is either explicitly passed as a parameter of the requests, or implicitly propagated by ORB, among the related transactional objects. The Control object is used in obtaining Terminator and Coordinator objects. Transactional client uses the Terminator to abort or to commit the transaction. Coordinator provides an interface for transactional objects to participate in two-phase-commit protocol. Transactional client sends requests to transactional objects. When a request is issued to a transactional object the transaction context associated with the invoking thread is automatically propagated to the thread executing the method of target object. A transactional object is the one that supports transaction primitives as defined by the standard. After the computations involved in the transaction have been completed, the transactional client uses the Current pseudo object to request that the changes be committed. OTS commits the transaction using 2PC protocol wherein a series of requests are issued to the registered resources. Thus ORB provides the atomicity of distributed transactions.

In addition to the above usage of OTS, in METUFlow OTS implementation, a method is added to the Coordinator object to handle xor_parallel block which requires one and only one task to commit, for the commitment of the block.

10. Correctness Issues in METUFlow

Since workflows are long running activities, having the transactions to commit within the scope of a workflow instance is an accepted practice. Thus the data modified by these transactions becomes accessible to the rest of the world which may cause inconsistencies. The problem is further complicated by the transactions that are compensated. Yet many scenarios in the operation of a workflow system require the preservation of data consistency of at least some data items.

It is possible to classify the data consistency problems involved into three categories:

1. Data inconsistency problems involving a single site,
2. Data inconsistency problems involving more than one site,
3. Data inconsistency problems due to compensation.

As an example to the problems of first category consider an *Order Processing* workflow in a manufacturing enterprise. In the processing of the *Order Processing* workflow raw material stock is checked through a task to see whether there is enough raw material in the stock to process the order. If not, the missing raw materials are ordered from external vendors. Yet later in the process when the actual manufacturing is to start for this workflow instance there might not be enough raw material in the stock to process this order, because a concurrently running instance of the same or other workflows might have updated the stock. Of course, executing all these tasks within the scope of a transaction might have solved these problems but workflow systems are there to prevent the inefficiency of long running transactions.

An example to the data inconsistency problems involving more than one site is as follows: Consider the *Withdraw-Deposit* workflow of a bank involving two branches as shown in Figure 10.1. *Withdraw* task withdraws the given amount of money from an account at the first branch, and the *Deposit* adds this amount to another account at the second branch. To preserve data consistency, no other task accessing the same account in any of the involved branches should go in between these two tasks. For example, consider an *Audit* workflow which checks the balance of these accounts. If *Withdraw-Deposit* tasks and tasks of the *Audit* workflow are interleaved incorrectly as depicted in Figure 10.1, *Audit* misses the money being transferred between the two accounts.

As indicated in the literature [**?**, **?**], early exposure of uncommitted data is essential in the realm of long-duration and nested transactions such as workflows. Since the tasks of a workflow are the grain of interleaving, and intermediate results are exposed, undo operations can no longer use the before-images. In case of failures, compensating tasks may have to be used to semantically undo the effects of committed tasks. The third type of problem occurs when a committed task after disclosing its updates to the outside world is

compensated. Tasks can be affected by the data disclosed by a previously committed task, in other words, their computations can be invalidated after the compensation of a task that they depend on.

In the following sections the solutions brought to these problems in METUFlow will be described.

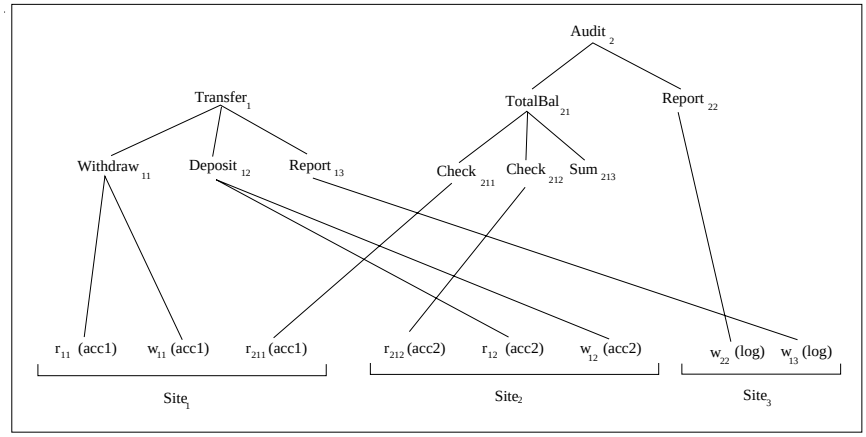


Fig. 10.1. Concurrency control problem of workflows at multiple sites

10.1 Concurrency Control in METUFlow

Data consistency can be violated by improper interleaving of concurrently executing workflows as discussed in the previous section. Also, such inconsistencies can occur due to improper interleaving of concurrently executing workflows and local transactions. Such interleavings must be prevented to ensure data consistency in WFMSs. In this section we introduce the "sphere of isolation" concept for the correctness of concurrently executing workflows. In achieving this goal we aim at increasing concurrency. Our starting point is to exploit the available semantics in workflow specification. How this semantic knowledge is extracted and usage of this knowledge to preserve data consistency are provided in the following.

10.1.1 Spheres of Isolation. We define a sphere of isolation to be the set of tasks that have data-flow and also serial control-flow dependencies among them. We claim that the workflow correctness can be provided by identifying the spheres of isolation in a workflow system automatically from the data

and serial control-flow dependency information obtained from the workflow specification.

If at least one of the output parameters of a task is mapped to an input parameter of the second task, we say that there is a *data-flow dependency* between these two tasks. There is a *serial control-flow dependency* between two tasks, if one of them is *begin on commit dependent* on the other one (See Section 3.1), i.e., first task can begin only after the commitment of second task. Two tasks belong to same *sphere of isolation* (denoted as Π_i^j, which means j^{th} sphere of i^{th} workflow) if there are *data-flow* and *serial control-flow dependency* between them. For example, the task checking raw material stock and stock update task in the manufacturing example belong to the same *sphere of isolation*. In our banking example depicted in Figure 10.1, $Withdraw_{11}$ and $Deposit_{12}$ tasks belong to the same *sphere of isolation*. Yet since there is no *data-flow dependency* between $Report_{13}$ and other two, $Report_{13}$ belongs to a different *sphere of isolation*. The complete list of *spheres of isolation* for Figure 10.1 is as follows: $\Pi_1^1 = \{Withdraw_{11}, Deposit_{12}\}$, $\Pi_1^2 = \{Report_{13}\}$, $\Pi_2^1 = \{Check_{211}, Check_{212}, Sum_{213}\}$, $\Pi_2^2 = \{Report_{22}\}$.

The point we want to make over here is the following: Since isolation of a whole workflow execution is unacceptable because of performance reasons, we want to discover smaller units of isolation. Since individual tasks of a workflow are isolated by local Resource Managers' concurrency controllers, our main concern is to observe data dependencies between these individual tasks and preserve these dependencies when required. Since we consider individual tasks as black boxes the only way of observing data dependencies between them is to check the *serial control-flow* and *data-flow dependencies* between them. Since these dependencies are available at design time, *spheres of isolation* can be determined automatically. Our notion of workflow correctness is based on the isolation of these spheres, i.e., if the tasks of a *sphere of isolation* execute at a single site, they are executed within the scope of a single transaction or if they execute at multiple sites their serialization order must be compatible at these sites. Note that, tasks which belong to different *spheres of isolation* may have incompatible serialization orders at multiple sites without violating the workflow correctness.

For example, since $Withdraw_{11}$ and $Deposit_{12}$ in Figure 10.1 belong to the same *sphere of isolation* their serialization order must be compatible at every site that they have executed, that is, $Site_1$ and $Site_2$. So, either $Withdraw_{11}$ must be serialized after $Check_{211}$ at $Site_1$ or $Deposit_{12}$ must be serialized before $Check_{212}$ at $Site_2$. Note that *Report* tasks can be serialized in any order, since they do not affect the correct execution of other tasks. So, for example $Report_{13}$ should not necessarily have a consistent serialization order with $Wihdraw_{11}$ and $Deposit_{12}$ for the correctness.

10.1.2 A Correctness Theory for Workflows. A formal presentation of *sphere of isolation* and a correctness theory developed to express the ideas introduced in the previous section more precisely is given in [**?**]. Note that the

theory introduced in [?] is motivated by the theoretical framework provided in [?] for nested transactions in multidatabases.

In this theory, an execution history of workflows is modelled by assuming an imaginary root (OMNI) for all submitted workflows. *Execution history of workflows* is a tree on tasks and $\rightarrow$ is a irreflexive and antisymmetric relation on the nodes of the tree. Actually, $\rightarrow$ is the ordering requirements on the leaf nodes due to execution order of conflicting data manipulation operations. Ordering imposed by leaf nodes are delegated to upper nodes in the hierarchy using the following axioms for any tasks t_i and t_j:

i. transitivity: if $t_i \rightarrow t_j$ and $t_j \rightarrow t_k$ then $t_i \rightarrow t_k$

ii. delegation: if $t_i \rightarrow t_j$ and

a. if $parent(t_j) \notin ancestors(t_i)$ then $t_i \rightarrow parent(t_j)$

b. if $parent(t_i) \notin ancestors(t_j)$ then $parent(t_i) \rightarrow t_j$. □

Within this theoretical framework, the correctness of a *sphere of isolation* can be checked and enforced by keeping its tasks under the same parent whereas unrelated parts of the workflow can be executed freely by making them the children of independent parents.

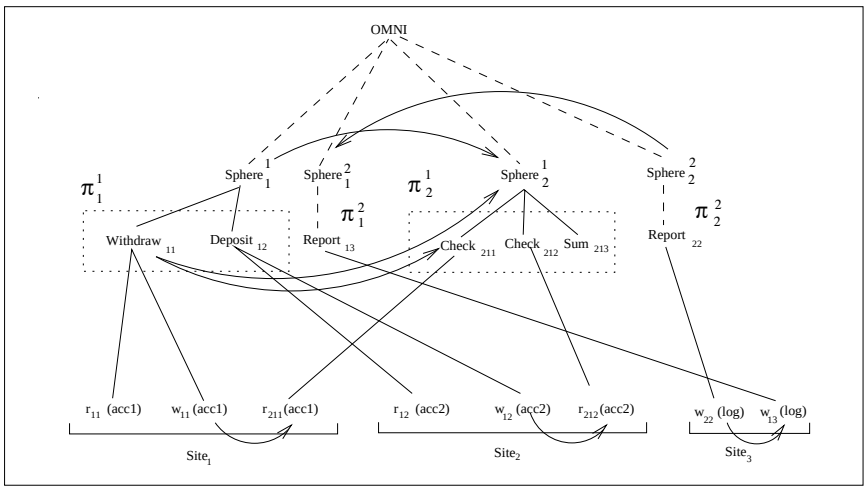


Fig. 10.2. Example Execution History

For example, consider the execution history in Figure 10.2 as a continuation of the example in Figure 10.1. *Spheres of isolation* are depicted within the dotted rectangles in the Figure 10.2. Parents of Π_1^1 and Π_1^2 are differentiated and a *virtual parent* for the elements of Π_1^2 is created and it is denoted as

$Sphere_1^2$. Similarly, $Sphere_2^2$ is created for the elements of Π_2^2 and the parent of the tasks of Π_1^1 is renamed as $Sphere_1^1$ and the parent of the tasks of Π_2^1 is renamed as $Sphere_2^1$. Since $Withdraw_{11}$ and $Check_{211}$ have issued conflicting data manipulation operations on $acc1$ they are ordered as $Withdraw_{11} \rightarrow Check_{211}$ at $Site_1$. Also $Deposit_{12}$ and $Check_{212}$ are ordered as $Deposit_{12} \rightarrow Check_{212}$. Since, $Withdraw_{11}$ and $Check_{211}$ are ordered as $Withdraw_{11} \rightarrow Check_{211}$, $Withdraw_{11}$ and $Sphere_2^1$ (which is $parent(Check_{211})$) are ordered as $Withdraw_{11} \rightarrow Sphere_2^1$ (from Axiom ii.a above). Some of the delegated orderings are not shown in Figure 10.2 for the sake of simplicity. By applying the delegation axiom repeatedly, the following order is obtained between different *spheres of isolation*: $\{Sphere_1^1 \rightarrow Sphere_2^1,\ Sphere_2^2 \rightarrow Sphere_1^2\}$. This execution is serializable and correct from the application point of view. Observe that, since $Report_{13}$ belongs to a different *sphere of isolation* (Π_1^2), its inconsistent serialization order with $Withdraw_{11}$ and $Deposit_{12}$ does not affect the correct execution of the workflow.

10.1.3 Implementation Issues. As can be seen from the discussion presented above, the spheres of isolation in a workflow can be identified and we claim that correctness measures can be applied on the basis of spheres of isolation. In [**?**] we present *Nested Tickets (NT)* technique to provide for the correctness of concurrently executing nested tasks of workflow systems, based on spheres of isolation. The main idea of NT technique is to give tickets to spheres and tasks. The NT technique makes the execution order of all tasks of a sphere of isolation to be consistent at all related sites. In other words, the consistency of serialization order of the tasks of a sphere of isolation is provided by guaranteeing them to be serialized in the order of their ticket numbers.

10.2 Future Considerations for Concurrency Control in METUFlow

Currently, we are in the process of developing a more efficient correctness notion for workflows exploiting workflow and task semantics in terms of semantic dependencies between tasks [**?**]. The concurrency control mechanism based on this semantic information will guarantee that only the task interleavings that preserve the correctness are allowed.

Another issue we are currently addressing is to bound the effects of compensation on the tasks that are affected by accessing data which is externalized by the compensated tasks [**?**]. Our primary goal is to analyze the implications of the compensation on the correctness of concurrent executions using task and workflow semantics and to provide correct executions with respect to compensation. We plan to present the user a mechanism with a flexibility of isolation levels that should be respected very much like the isolation levels of SQL-92. Hence critical tasks will have chance to choose not reading subject to compensation data.

11. Conclusions

Currently the first prototype of METUFlow is operational. The system attempted to bring solutions to the following problems of the workflow systems:

1. Scalability and adaptibility through distributed scheduling, history and worklist management.
2. Handling of invoked applications on distributed heterogenous environments through CORBA.
3. Supporting mobile users through the Java based Web interfaces.

With the future versions of METUFlow, we plan to attempt the following problems:

1. Deadlock and reachability analysis of workflow specifications [**?**].
2. Handling dynamic changes within workflow specification which boils down to changing guard expression of events dynamically in the system.
3. We also aim at an adaptable workflow system that will incorporate the functionality and therefore complexity only when it is actually needed. The notion of adaptability implies that the engine will run in different platforms ranging from a small set of PCs for minor administration tasks, to clusters of high end workstations connected through a wide area network for enterprise wide system.

Index