Design and Implementation of a Distributed Workflow Enactment Service *

Esin Gokkoca, Mehmet Altinel, Ibrahim Cingil, E.Nesime Tatbul, 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

Abstract

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

In this paper we present the design and implementation of a distributed workflow enactment service based on the work given in [12]. Yet by starting with a block structured workflow specification language we avoid the very general set of dependencies and their related problems. In this way it is possible to present a simple algorithm for distributed scheduling of process instances. Further benefits of the approach are the ease in testing and debugging the system and execution efficiency through reduced number of messages.

1 Introduction

Workflows are activities involving the coordinated execution of multiple tasks performed by different processing entities. Since they execute in distributed heterogeneous environments involving a variety of human and system tasks, distributed scheduling of workflows is essential.

Distributed scheduling of workflows has been addressed in [1, 3, 12]. In [1], a distributed workflow system is proposed based on persistent message queues where the authors assume that the processes are well-formed, i.e., they do not have cycles or dependencies that may compromise their execution. The authors further assume that the data flow follows the ordering imposed by the control flow to avoid race conditions. [3] proposes INCAs for distributed workflow management. In this model, each execution of a process is associated with an Information Carrier (INCA), which is an object that contains all the necessary information for the execution as well as propagation of the object among the relevant processing nodes.

In this paper we describe the design and implementation of a distributed workflow enactment service based on the work presented in [12]. Workflow enactment service 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. Workflow enactment service maintains control data and workflow relevant data and uses them to evaluate entry and exit criteria for activity steps. It is also responsible for supervisory actions of control, administration and audit [7].

The approach taken in [11, 12] for distributed scheduling of workflow executions is based on the following observations and mechanisms: The main activity of a workflow is organization and coordination of tasks which might be dependent on each other by their states. For example start of one task may depend on the commitment of another task. Since state changes of the tasks are represented as event generations, state dependencies can be defined through event ordering. Therefore controlling the occurrences of events provides the coordination of the tasks. In other words, intertask dependencies are represented by

* 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 Motorola (USA) and by Sevgi Holding (Turkey).

the event dependencies. To make distributed execution of workflow computations possible, occurrences of events are not controlled in a central scheduler. Instead, each event is made responsible for controlling its execution to decide on the right time to occur. Required information for this operation is obtained from the dependency expressions after a refinement process which is termed as guard compilation. Guards are temporal expressions defined on events and occurrences of events are permitted only if their guards are true. When an event happens, messages announcing its occurrence are sent to other related events. Tasks are interfaced to the system through agents. An agent embodies a coarse description of the task, including states and transitions. In addition, an actor is instantiated for each event. The actor for an event maintains its current guard and manages communication of necessary messages. Guards of events are determined by generating all possible computations relevant to each dependency. In evaluating a guard, the intrinsic attributes of events must be taken into account. The following event attributes are defined in [12]: (a) Normal events that are delayable and rejectable (e.g. commit), (b) Inevitable events that are delayable and nonrejectable, (c) Immediate events that are nondelayable and nonrejectable (e.g. abort), and (d) Triggerable events that are forcible (e.g. start). The event attributes are taken into account during guard generation [12].

The following points have been notified with this approach:

1. In [2], guard generation process is said to run into combinatorial explosion. The proposed process first determines all possible paths for a given dependency. There are n! number of paths for a dependency involving n events. Later in [12], it is proposed that by relaxing the past and the future, the guards can be generated symbolically without the need for determining all possible paths.

2. The execution mechanism is based on message exchange between actors since guards on events require notification messages to assimilate the event executions. This in turn might arise potential race conditions and deadlocks. For example there could be two events waiting for the occurrence of each other, resulting in a deadlock. It is therefore essential to preprocess the guards so as to detect and resolve the potential deadlocks through promissory messages [12].

3. The guards may contain events that refer to the future, however in actual execution we do not have the luxury of looking into the future. In [12], various heuristics are suggested, yet the completeness of the suggested heuristics is left as a future work.

In order to confine the theory presented in [12] to a manageable practical implementation, we started by designing a block structured procedural workflow specification language. In this way it becomes possible to express the workflow specification with a well-defined set of dependencies. We show that these dependencies produce simple guard expressions which in turn makes it possible to give a simple algorithm to generate the guards from the specification language by also taking the event attributes into account.

The block structured nature of the specification language makes it also possible to locate and handle the deadlocks and race conditions without the need for preprocessing the specification as explained in Section 3 .

Furthermore, in our workflow specification language, because of its well-defined semantics, the references to the future are known at compile time and can thus be easily handled by a special software module (a modified 2PC protocol implementation) as explained in section 3.2.

The paper is organized as follows: In Section 2 the process model and the semantics of different block types are presented. In Section 3, first a formal definition of the blocks are given in ACTA formalism which is then mapped to two dependencies given in [8]. We then demonstrate the guard simplification process and explain guard construction through an example. Section 4 describes the distributed execution environment. Finally conclusions are presented in Section 5.

2 The Process Model

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 blocks are conceptual activities which are present only to specify the ordering and the dependencies between activities.

We have seven types of blocks, namely, serial, and_parallel, or_parallel, xor_parallel, contingency, conditional and iterative blocks. The following definitions describe the semantics of these block types where B stands for a block, A for an activity and T for a task.

Definition 1 B = (A_1 ; A_2 ; A_3 ; .....; A_n), where B is a serial block. 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, then committed activities in the block are compensated in the reverse order.

Definition 2 B = (A_1 & A_2 & ..... & A_n), where B is an and_parallel block. 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, then committed activities in the block are compensated in parallel.

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

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

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

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

Definition 7 B = (condition;$A_1; A_2; .....; A_n$), where B is an iterative block. The iterative block B is similar to a 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. If B aborts, then for all iterations, committed activities in the block are compensated in the reverse order.

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

Definition 9 T = T_u, where T_u is the undo task of task T. The undo task T_u of T starts if T fails.

In addition to activities, there is 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 int patient_id;
 var STRING result1, result2;
 var int  blood_test_type_list, roentgen_list;

 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);
 }
}
```

In MFDL, we have used 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 1. The significant events in our model are start, commit and abort. The event attributes of these tasks are shown in Table 1.

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

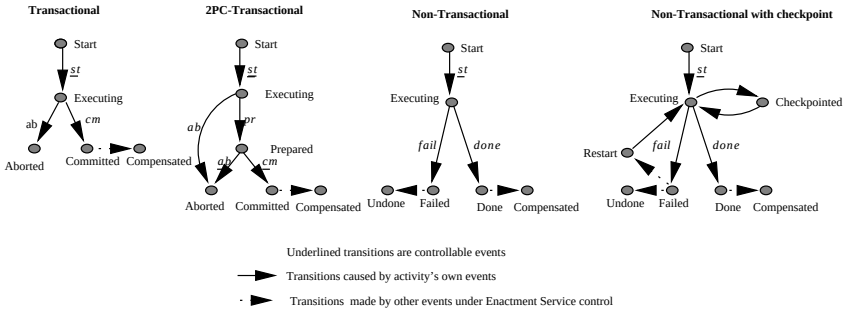

Figure 1. Typical task structures

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

Table 1. Event Attributes

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 [6, 4]. 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 and activities 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 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.

3 Guard Construction

Our primary aim is to create a distributed execution environment for the activities in our workflow system. Since our sytem is distributed on the basis of activities, each activity should know when its significant events like start, abort and commit should occur without consulting to a top-level central decision mechanism. For this purpose, we use temporal expressions which define the conditions under which an event should occur, called guards [12].

In this section, first the semantics of the block types are defined using ACTA formalism. We then show that the two dependencies provided in [8] are adequate to express the specified block semantics and result in simple guard expressions.

3.1 Semantics of the Block Types using ACTA Formalism

We use the ACTA formalism [5] with slight modifications to express the semantics of block types[1]. The ACTA dependencies used for this purpose are presented in the following:

Let t_i and t_j be two transactions.

- **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.

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

- **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 dependencies. These dependencies have an additional argument which is "condition". For example, we express conditional begin dependency as BD(C). If condition C is true, then BD holds, else it does not hold.

Using these dependencies, we can formally restate semantics of block types defined in the previous section.

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 $\leq$i<n
- B CD A_n
- B AD A_i ,1 $\leq$ i $\leq$n

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

- A_i BD B ,1 $\leq$ i $\leq$n
- B AD A_i ,1 $\leq$ i $\leq$n
- $\forall$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 $\leq$ i $\leq$n
- $\exists$i (B CD A_i) $\wedge$ ($\forall$j((B CD A_j) $\vee$ (B CAD A_j))), j$\neq$i
- $\forall$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 $\leq$ i $\leq$n
- $\exists$i (B CD A_i) $\wedge$ ($\forall$j(A_j ACD A_i)), i$\neq$j
- $\forall$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 $\leq$ i<n
- B CD A_i, 1 $\leq$i$\leq$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($\neg$C) B
- B CD(C) A_1
- B CD($\neg$C) A_2
- B AD(C) A_1
- B AD($\neg$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 $\leq$i<n
- B CD($\neg$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 we will use the following two primitives [8, 2] 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 we use in specifying 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 [2, 12]:

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.

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

Table 2. Guards corresponding to the dependency set

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 [12]:

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

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

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 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 2. 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 2.

If we summarize, by starting from 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 dependency produces straightforward guards for events. This 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 2.

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

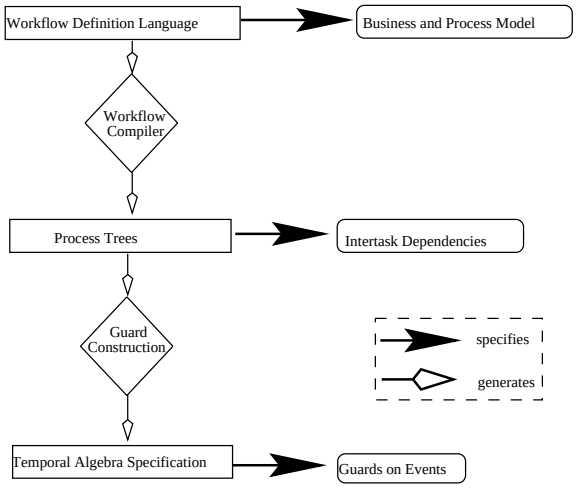

Figure 2. Guard generation process

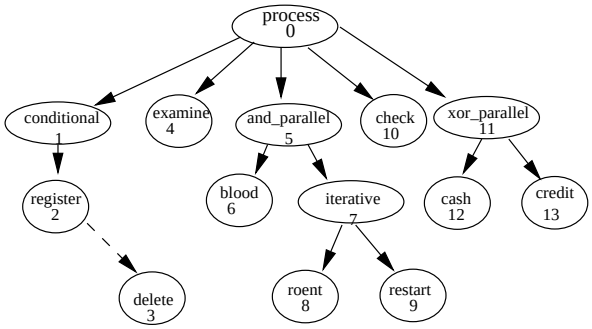

Figure 3. Process tree of the example MFDL

symbols. This tree explicitly shows the dependencies between the activities of the workflow. In fact, with the Table 2 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.

In Figure 3, 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 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 2,

G($D_1, 3_{st}$) = $\Box 0_{ab}$

G($D_2, 3_{st}$) = $\Box 2_{cm}$

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	$\Box 11_{st}$		$\Box 12_{cm}$	$\Box 12_{ab}$	

Table 3. Guards of the example workflow definition

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, 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 [11] by resurrecting a guard under appropriate conditions. Ours is a practical implementation of this formal concept.

The Table 3 shows the start, abort and commit guards for all the nodes of the example process tree given in Figure 3.

It should be noted that in Table 3, 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, 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. 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 commit. In this case, the abort and commit guards are not constructed any more for the immediate child nodes.

4 The Execution Environment

After the guards are constructed, an environment in which these guards are evaluated through the event occurrence messages they receive is created. Our approach associates a guard handler with each activity instance which contains the guard expressions for the significant 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. The task handler acts as an interface between the activity instance and its guard handler. A guard handler provides the message flow between the activity'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 activity to occur.

Each node in the process tree is implemented as a CORBA [13, 9] object with an interface for the guard handler to receive and send messages, as shown in Figure 4. 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();
               ...
```

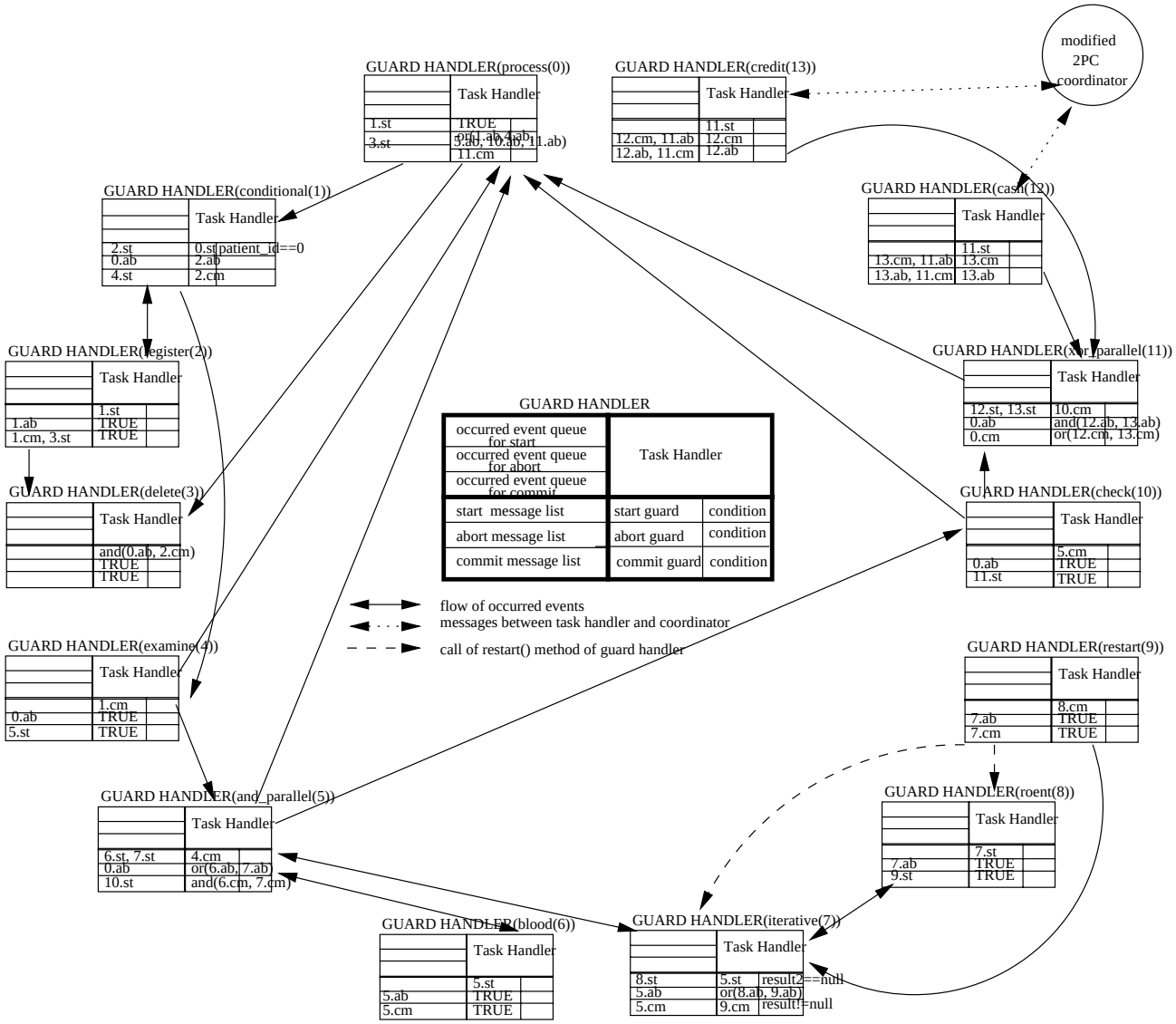

Figure 4. Execution environment of objects of Guard Handler

```
                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 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.

When block abstraction is used during execution as shown in Figure 6, 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 6.

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

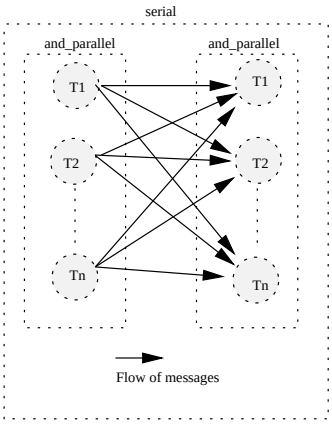

Figure 5. Environment of objects without block abstraction

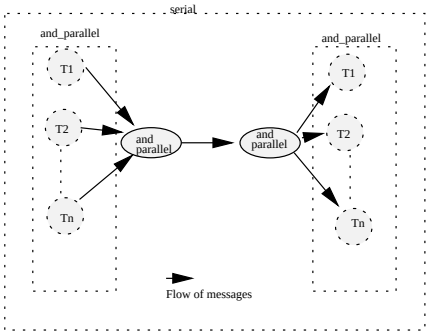

Figure 6. Environment of objects with block abstraction

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 4 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 disabling 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.

When it comes to maintaining workflow relevant data and history management, note that data about workflow events arise at different sites where workflow activities are executed. Currently in METUFlow, the history of each activity instance and each workflow relevant data are implemented as CORBA objects.

All the information about an activity instance is stored in a history object to be used for monitoring and data mining purposes. These objects are linked through their object identifiers according to the process tree. Each activity instance is responsible from 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. Using these links a history tree is constructed when necessary. In a workflow relevant data object, the information about this data like the time it is created, its initial value, and its versions is stored. All the activities can reach these objects through CORBA without having the knowledge of at which site they are created.

5 Conclusions

In this paper a distributed workflow enactment service based on the work presented in [11, 12] is described. Main contributions of the paper are as follows:

- A block structured procedural workflow specification language is presented. This made it possible to avoid the very general set of dependencies and their related problems during distributed scheduling of process instances. More specifically the following advantages are obtained in this respect:
  - A very simple mechanism for guard construction and execution is provided.
  - Race conditions and deadlocks are avoided and references to future are detected at the compile time and are handled easily.
  - The abstraction provided by the block structures is exploited during execution and this abstraction reduces the number of messages necessary for the distributed enactment service.

- It should be noted that MFDL has the expressive power of ECA rules since it includes such a construct through its conditional block.
- 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.
- Distributed nature of the enactment service provides for failure resiliency; a failed node effects only those nodes waiting a message from it; execution can continue in other nodes.
- History and workflow relevant data management are handled in a distributed fashion.
- And finally, ease in testing and debugging is provided. As noted in [10], 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. Our approach prevents these problems.

The futurework for this project includes incorporating temporal dependencies and concurrency control dependencies into guards.

References

[1] G. Alonso, C. Mohan, R. Gunthor, D. Agrawal, A. El Abbadi, and M. Kamath, *" Exotica/FMQM:A Persistent Message-Based Architecture for Distributed Workflow Management"*, in IFIP WG8.1 Working Conference on Information System Development for Decentralized Organizations, Norway, 1995.

[2] P. Attie, M. Singh, A. Sheth and M. Rusinkiewicz, *"Specifying and Enforcing Intertask Dependencies"*, in Proceedings of the 19th International Conference on Very Large Data Bases (VLDB'93), 1993.

[3] D. Barbara, S. Mehrota, and M. Rusinkiewicz, *"INCAs: Managing Dynamic Workflows in Distributed Environments, in Journal of Database Management"* , Vol.7, No.1, Winter 1996.

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

[5] P. K. Chrysanthis and K. Ramamritham, *"ACTA: The SAGA Continues"*, in Database Transaction Models for Advanced Applications, Morgan Kaufmann Publishers 1992, edited by A. K. Elmagarmid.

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

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

[8] J. Klein, *"Advanced Rule Driven Transaction Management"* in Proceedings of the IEEE COMPCON, 1991.

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

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

[11] M. Singh, *"Synthesizing Distributed Constrained Events from Transactional Workflow Specifications"*, in Proceedings of the 12th International Conference on Data Engineering (ICDE'96), New Orleans, February 1996.

[12] M. Singh, *"Distributed Scheduling of Workflow Computations"*, Technical Report, Department of Computer Science, North Carolina State University, 1996.

[13] R. M. Soley (ed.) and C. M. Stone, *"Object Management Architecture Guide"*, Third Edition, John Wiley & Sons, 1995.