CHAPTER V
CHAPTER V

Implementation Details

In this chapter we discuss the mapping of the conceptual model developed for the document structure to an object-oriented database and give detailed algorithms for generation of the deadline table for presenting a document having time dependencies between its components.

Converting document structure into classes

The document structure is translated into an appropriate set of classes for an object-oriented database. We are not assuming any specific database because the lack of a formal model. The well-defined concepts like, tuple, set, generalization, aggregation etc. are used. The classes defined in this chapter use the abstract and generic classes which have been defined in chapter II on modeling of multimedia data.

fig. 5.1
In order to map the document Dept (given in fig. 5.1) to an object-oriented database, the following set of classes are defined:

**class Dept** subclass of root
(compo-Lab-ref, set-ref, tuple<intro, professorlist>)

attributes
{
    dept_name : string;
    dept_off-Rno : integer;
    enrol_stren : integer;
    emp_stren : integer;
    dept_chairman;
}

methods
{
    enrol_student();
}

**class intro** subclass of temporal
(compo-history-expI-ref, set-ref, tuple<history, expl1>)

attributes,
{
    composer: string
    photograph: image;
    list_of_proj : set,
    list_of_courses : set;
}
methods
{
    get_proj_expl;
    get_course_list;
}
class history subclass of text
(assem-intro-ref, atomic-ref, intro)
attributes
{
    dt_estab : date;
    founder_chairman : string
}
methods
{
    display( );
}
class expl, subclass of audio
(assem-intro-ref, atomic-ref, intro)
attributes
{
    narration : audio;
}
methods
{
    play( );
    volume( );
}
class proflist subclass of spatial & temporal
(compo-profdetail-ref, set-ref, profdetails)
attributes
{
    no_of_prof : integer;
}
methods
{
    get_ind_prof( );
    get_seniority_list( );
    get_prof_in_proj( );
}

class profdetails subclass of temporal and spatial
(assem-proflist-ref, atomic-ref, proflist)
(compo-expl2-name-resume-pic-ref, set-ref, tuple<expl2, resume, name, pic>)
attributes
{
    list_of_students : integer;
    res_address : tuple;
}
methods
{
    get_resume( );
    get_stud_list( );
}
**class name** subclass of text

(assem-ref-profdetails-ref, atomic-ref, profdetails)

attributes

{
    first : string;
    last : string;
}

methods

{
    abbrev_name( ),
}

class expl; subclass of audio

(assem-ref-profdetails-ref, atomic-ref, profdetails)

attributes

{
    profile : audio;
}

methods

{
    record_prof( ),
}
class resume subclass of text
(assem-ref-profdetails-ref, atomic-ref, profdetails)
attributes
{
  edu_qual : text;
  dt_of_joining : date;
  proj_assoc : set
  publ_list : set;
}
methods
{
  list_publ( );
}

class pic subclass of image
(assem-ref-profdetails-ref, atomic-ref, profdetails)
attributes
{
  click:don date
  photo_studio_no : integer;
}
methods
{
  display_stamp_size( );
}
**Algorithms for creating templist, nodelist and deadline table**

We now give description of various structures used to create templist, nodelist and finally the sorted deadline table.

The document is organized as n-ary tree. Each node (called t\_node) contains object-identifier, called objid which identifies the object uniquely. For the algorithms in this section we store the synchronization specifications, namely, offset, duration and t\_rel (temporal relation) in the nodes of the tree besides objid. Besides this each node has two pointer, child - pointing to the leftmost(first) child and bros - pointing to its next sibling (on right). The child and bros are pointers to type t\_node. The structure t\_node which is a node of the document tree is:

```c
{
   id of the type integer;
   offset of the type integer;
   dur of the type integer;
   t\_rel of the type character string;
   child and bros pointers to type t\_node
}
```

The syn\_node is a node of the templist. The detailed structure of the syn\_node is described in detail in fig.3.16. It stores synchronization relation between a pair of nodes, one node acting as the source and second as destination. The synchronization information between the two nodes is stored as trig which a 2x2 array of rec. Because of the linked data structure each node contains a field next which is pointer to the next node. The structure of the syn\_node is:
{  
  src and dest of type integer;
  trig is a[2][2] array of rec;
  next is pointer to type syn_node;
}

The rec is a part of type of element of trig array. trig[i,j] described in syn_node has two components, identification of source or destination and the associated event, i.e. begin or end. Therefore structure of rec is defined as:
{  
  id of type integer;
  event of type integer;
}

For implementation purposes we store the nodelist described in section 4.2.1.1 as an array, instead of linked list. Nodelist consists of n structures, where n is the total number of nodes in the document. Each of this structure is called a block_array. The detailed structure of block_array is given in fig 4.6. The block array consists of a node id which is the source of synchronization and all other nodes(i.e. destinations) which have some kind of relation with the former. The destination node ids have been organized as 2x2 array of next. next is a pointer to the type l_node. The structure of block_array and l_node are given below:
{  
  id of type integer;
  array [2][2] of next;
}
The structure of \texttt{l\_node}
{
    id of type integer;
    next of the type pointer to \texttt{l\_node};
}

In order to carry out breadth-first scanning of the document tree we require a queue structure. The structure of element of a \texttt{queue} is given below. The queue is implemented as linked list. \texttt{Qpointer} is pointer an element of queue.
{
    entry of type pointer to \texttt{t\_node};
    next of type \texttt{Qpointer};
}

The table is our final output which stores the deadline table. The number of entries in the table will be equal to the number of nodes(n) in the document tree. The structure of \texttt{table} and \texttt{table\_row} which is an entry of the table are described next.

{
    table\_row[n];
}

The structure of \texttt{table\_row} is
{
    int id;
    char leaf\_node;
    int start\_time, end\_time, another\_end\_time;
}
Below we give detailed algorithms to create templist, nodelist and the deadline table. These have been implemented in C.

To get the deadline table the algorithm goes through three main phases. The first step is to create a synchronization list which consists of nodes of type syn_node. This is accomplished by algorithm traverse-tree. The intermediate result of this phase, i.e. the list of syn_nodes called templist is displayed using algo displaysynchlist. The next is to create nodes of the type block_array. This is done by algorithm createblock_array which uses the templist as the input. The result of algorithm reateblock_array is displayed by displayblock_array. Finally deadline table is created by createdeadline table which uses the list of block_array as well as the values of duration and offset from the original document tree as inputs. For carrying out the above three steps in the main algorithm invokes create-doc to create a document tree structure with required attributes.

algo main
create_doc();/*to create the document tree*/
traverse_tree(root);/*creates templist by traversing document tree already created*/
displaysynchlist ( start);/*displays templist*/
createblock_array ( start );/*creates nodelist from templist*/
displayblock_array();/*displays nodelist*/
createdeadline_table;/*creates deadline table taking synchronization constraints from nodelist and other information from the document tree*/
end main
algorithm create_doc

algo create_doc ()
/*create_doc creates a document structure by setting values of id, temporal relation, 
offset and duration. ptr is of the type pointer to t_node, id gives the object 
identification, t_rel gives the temporal relation stored in the node and is stored as 
array of char, dur gives duration of display of the particular object and is stored as 
dur. This algorithm returns a pointer to type t_node*/
/*create root node*/
set ptr->child to Nil;
set ptr->bros to Nil;
set the values of id, t_rel and dur
root = ptr;
add_node_in_tree(root);
return root;
end create_node

algo add_node_in_tree(root)
/* this algorithm creates the rest of the document tree as per user requirements. ptr 
and temp are pointers to type t_node, n and i are integers*/
set temp = root;
npu t n /*input number of children of temp as variable n*/
f n > 0
{ temp = create_node();/*a new node is created*/
  set p->child = temp; set p=p->child;
  if n-1 > 0
  {
    set ptr = p;
  }
for i = 1 to n
{
    temp = create_node();
    set p->bros = temp;
    set p = p->bros;
}
}
while ptr != Nil
{
    add_node_in_tree(ptr); /*recursive call to the algo*/
    set ptr = ptr->bros;
}
end add_node_in_tree

The create_doc algo results in a document tree organized as given in fig.5.2. We input
the objid and synchronization specifications. id is objid, o is offset, d is duration and
t_rel is temporal relation.
The algorithm traverse_tree carries out breadth first scanning of the tree created by create_doc with root node = root(as given by create_doc and creates a list of syn_nodes called templist. The document has been stored as n-ary tree; each node has two pointers; the first pointer(child) points to the leftmost child and the second pointer (bros) points to the next sibling. Starting from the root node the tree is traversed from Left to Right at each level, and the temporal templates get instantiated accordingly. For each parent node we assume that the relation it has with one of its children(i.e. the child with earliest start time) is that of parallel start. Thus the template for parallel start is instantiated. For all the children from left to right with ith child as the source and i+1th child as the destination the particular template as given by the temporal relation is instantiated. Another temporal relation parfinish is required between the child with latest finish time and the parent. So parfinish template is instantiated after the last child of the parent is visited (before the next sibling of the parent is visited). The first procedure to be called from traverse-tree is visit with t as the parameter and first of all parstart template is instantiated by procedure createsyncblock invoked from visit. Control comes back to traverse-tree. In the next step, templates for temporal relations between siblings of the next level are generated. For the last child at a particular level parend template is generated by invoking createsyncblock via visit, with src as the last sibling and as the parent. Each time after creating a node addnodelist is called which appends the latest node to the link list. The same process is repeated for all subtrees. Algo createsyncblock creates a node of the link list by instantiating the appropriate template as required by the t_rel field of the parent in case of siblings and parfinish or parstart in case of last child and first child respectively. The values of src, dest and tr are passed as parameters. This procedure sets the values of array trig. The array trig is 2x2 and each element is (id,event) pair. id denotes the id of source in the first column and destination in the
second column. If the value of event is 1 it implies begin and value 0 implies end. Tree traversal stops when the queue is empty and finally the list of nodes is displayed.

\textbf{algo traverse_tree}(root)

/*procedure traverse_tree traverses the tree like breadth first fashion and ultimately creates the templist which is a list of templates; p is pointer of type t_node and Qptr is pointer of type QPointer*/

set t = root

while t != Nil

{ 
    visit(t); /*visit() creates a node of templist*/
    if t->child != Nil

        { /*node is added to queue to accomplish Breadth-first search*/
            set Qptr=CreateQueNode(t->child);
            AppendInQue(Qptr);
        }

    t=t->bros;
}

if queue not empty

{ 
    p=Serve(); /*serve retrieves the node at the head of the queue*/
    traverse_tree(p); /*recursive call to algo*/
}

end traverse_tree
\textbf{algo visit}(t)

/*syn\_block is a node of templist. src and dest are integers which give the object ids of source and destination nodes between which the current synchblock is being created; p is of the type \texttt{t\_node}, i and j are integers; trig is a 2x2 array, fresh is a pointer to the type sync\_block*/

set fresh=Nil;
if(t->child!=Nil)
{
    set src=t->id;
    set p=t->child;
    set dest=p->id;
    set trig for \texttt{parstart}; /*relation between parent and the first child*/
    /*call CreateSyncBlock to create node of templist and return pointer fresh to the newly created node*/
    fresh=CreateSyncBlock(src,dest, 'parstart');
    AddNodeInList(fresh); /*add node in the templist*/
    while(p->bros!=Nil)
    {
        set src=p->id;
        set p=p->bros;
        set dest=p->id;
        set array trig for the temporal relation mentioned in the parent
        /*call CreateSyncBlock to create node of templist and return pointer fresh to the newly created node*/
        fresh=CreateSyncBlock(src,dest, t_.rel);
        AddNodeInList(fresh); /*add node in the templist*/
    }
    set src=p->id;
set dest=t->id;
set trig for paren
/*call CreateSyncBlock to create node of templist and return pointer fresh
to the newly created node*/
fresh=CreateSyncBlock(src,dest,'parend');
AddNodeInList(fresh);/*add node in the templist*/
}
end algo;

algo createSyncBlock(source,destination,t_rel)
/*this algorithm creates a new node to be added in the templist every time any of the
five templates given in fig. 3.14 is instantiated while traversing the tree and returns
pointer new1 to the visit procedure, new1 is of the type pointer to syn_bloc(which is
a node of templist), i and j are integers. This algorithm returns pointer to the type
syn_block*/
set new1->src=source;
set new1->dest=destination;
for(i=0;i<2;i++)
{
for(j=0;j<2;j++)
{
set new1->trig[i][j].id=t_rel[i][j].id;
set new1->trig[i][j].event=t_rel[i][j].event;
}
}
set new1->next=Nil;

151
return newl;
end algo;

algo AddNodeInList(fresh)
/*this algorithm adds node pointed to by fresh to the templist; last and start are
pointers to the type sync_block */
if templist is empty
{
  set start=fresh;
  set last=fresh;
}
else
{
  set last->next=fresh;
  set last = fresh;
}
end algo;

algo AppendInQue(ptr)
/*this algorithm adds node pointed to by ptr in the queue for breadth first traversal,
queue contains a list of pointers to the nodes to the document tree*/
if (front==Nil)
{
  set front=ptr;
  set rear=ptr;
}
else
set rear->next=ptr;
set rear to point to the latest node;
}
ptr->next=Nil;

algo Serve
/*this algorithm retrieves the first node in the queue and returns a pointer to the retrieved node, p is a pointer of the type Qpointer*/
if (front!=Nil)
{
set p=front;
set front=front->next;
if (front==Nil)
    set rear=Nil;
}
return(p->entry);
end algo

algo CreateQueNode(t)
/*this algorithm creates a node to be appended in queue and returns Qpointer which is a pointer to an element of queue*/
set q->entry=t;
set q->next=Nil;
return q.
end algo
**algo Displaysynchlist(start)**

/* this algorithm is used for displaying the templist, start points to the beginning of
the templist and sptr is of the same type as start*/

sptr=start

while sptr!=Nil

{  
    print sptr->src);  
    print sptr->dest);  
    print array sptr->tr;  
    sptr=sptr->next)  
}

The link list created by algorithm traverse_tree is of the type given below in fig.5.3:

![Diagram](image-url)  

*fig. 5.3*
**algorithm createblock_array**

This algorithm creates nodelist from the templist. This nodes list is used in the generation of deadline table. For each node we store a list of all other nodes which have any kind of synchronization relation with this node with identifier id. Each node of this list consists of an object id and a 2 x 2 array. The elements of these array are further pointers to link list. The first column of the array refers to the begin event of the object for which this node is created and the second column refers to the end event pertaining to this object. Alternately, we can say that the first row refers to such nodes whose beginnings are triggered by the node id (whether its beginning or end), and the second row refers to those nodes whose ends are triggered by node id (whether beginning or end). next refers to the next such node in the link list. This nodelist consists of as many number of nodes as the number of objects comprising the document. This process is carried out by scanning of the list created by algo treetraversal for each node id and setting the value of appropriate ptr[i,j] by src or dest id. Value of i is given by the value of row of array trig where the id appears and the value of column depends on the event associated with the id, i.e. for begin j is set to 1 and for end j is set to 2. Thus we get a list with a node corresponding to each id.

```
algo createblock_array(start)
/*this algorithm creates the nodelist from the templist, start is the input to this algorithm which is pointing to the start of the templist, n is the total number of elements in the list which is the number of nodes in the document tree*/
for n nodes do
{
    for(i=0; i<2; i++)
    {
    
155
for(j=0; j<2; j++)
{
    block_array[i][j].CNHead=Nil;
}
}

set sptr=start;
while sptr!=Nil ; sptr=sptr->next)
{
    for(i=0; i<2; i++)
    {
        for(j=0; j<2; j++)
        {
            if(sptr->tr[i][j].id==id) and (sptr->tr[i][j].event==0))
            {
                if(j = 0)/*in the first column*/
                    set freshid=sptr->src;
                else
                    {
                        if(j==1)/*in the second column*/
                            set freshid=sptr->dest;
                    }
            CNfresh=CreateCNode(freshid);/*a node for nodelist is created*/
            append the node in the appropriate link lis pointed to by last1 or last2 or
            last3 or last 4 depending on the value of event, i and j*/
            
}/*end of if*/
algodisplayblock_array(ptr)
/*This algorithm displays the nodelist, i, j, k are integers and the input is the pointer to
the start of templist*/
while not (end of nodelist)
{
    print ptr->id;
    set j = 1; /*for the first row*/
    /*print begin triggers*/
    for k = 1 to 2
    {
        set ptr=ptr->block_array[j][k].CNHead;
        while ptr != Nil
        {
            print ptr->id;
            set ptr=ptr->next;
        }
    }
    /*print end Trigger*/
    set j=2;
    for k = 1 to 2
    {

ptr=block_array[i].array[j][k].CNHead;
while ptr != Nil
{
    print ptr->id;
    set ptr=ptr->next;
}
}

end algo

The nodelist created by the above algorithm is of the type given in fig. 5.4 below:

![Diagram](image)

fig. 5.4
algorithm createdeadline_table

This algorithm forms the last phase of our implementation. In this phase the final deadline table is created from nodelist created by createblock_array algorithm. For creating the deadline table we traverse the tree in pre-order every time pushing a non-leaf node in a stack and popping the parent when its last child leafnode is processed. Again use the begin and the end triggers from the nodelist to update start/endtime of the destination nodes given by the ptr array in block_array. If an objid.begin is begin trigger for some nodes then the start times of all such nodes are set equal to the start time of this node(after taking care of the relative offsets). If objid.begin is end trigger of some nodes then the endtimes of all such nodes are set equal to start time of this node. Similar updations are done for objid.end. For every node its start time is set by some other node. For the leaf node we calculate its end time also. Besides this for the last leaf sibling the end time of parent is also set. And then another deadline is calculated. Therefore we have two routines, update_deadline_1 and update_deadline_2.

algo createdeadline_table
/*this algorithm creates the sorted deadline table which is input to the playout phase*/
traverse_tree_pre(root)
sort_table(table);/*sorts the deadline table in ascending order of time*/
print_table(table);/*prints the final deadline table*/

end algo
algo traverse_tree_pre(root)

/* This algorithm traverses the tree in preorder and takes the tree root ptr as the input parameter. While traversing it gives a call to procedure store_order_id() which stores the ptr id and related information in the array order_id[]. j is declared to be static since the procedure gives recursive calls which results in re-initialization of the variable. k, id and count are integers*/

set ptr = root
if (ptr != Nil)
{
    update_deadline_1 (ptr->id);  /*procedure to update start-times*/
    if ( ptr->child != Nil) /*not a leaf node push in stack */
        push_stack(ptr ->id );
    else /*for a leaf node*/
    {
        find index k which gives table entry for id
        set table[k].end_time = table[k].start_time +ptr->offset+ ptr->dur,
        update_deadline_2 ( ptr->id);  /*procedure to update endtimes*/
        if ( ptr ->bros == Nil)/*for the last child of the parent pop parent out*/
            { id = pop_stack( );
                /*update deadline of parent*/
                table[id].end_time = table[id].start_time+duration of parent;
            }
    }
}
store_order_id(ptr,j);  increment j by 1;
traverse_tree_tree(ptr->child);
traverse_tree_pre(ptr->bros);
}
end algo
algo store_order_id(ptr, j)
/* Algorithm store_order_id() stores the pointers to the nodes appearing in preorder in the tree. Array order_id[] is for used, so that the tree has not to traversed repeatedly. ptr is pointer to type t_node and j is an integer*/

order_id[j].id = ptr->id;
order_id[j].duration = ptr->dur;
if(ptr->child == Nil)
    order_id[j].ln = 'y';
else
    order_id[j].ln = 'n';
end algo

algo update_deadline_1(id)
/*this algorithm updates the start times of nodes according to the synchronization constraints given in individual nodes of the nodelist. The input is id, the identifier of the node of the document tree whose start time has to be set. It starts with the root. i and k are integers*/

find the node in the nodelist corresponding to the id
if (block_array[i].ptr[0][0] != Nil)/*id.beg is begin trigger for any node*/
    begin_begin(block_array[i].ptr[0][0],k);
if (block_array[i].ptr[1][0] != Nil)/*id.beg is end trigger for any node*/
    begin_end(block_array[i].ptr[1][0],k);
end algo
algo update_deadline_2(id)
/* Procedure update_deadline_2() updates end times of the destinations of nodes
synchronization as given in the nodelist*/
find the node in the nodelist corresponding to id;
if(block_array[i].ptr[1][1] != Nil)
    end_end(block_array[i].ptr[1][1],k);/*if id end is end trigger for any node*/
if (block_array[i].ptr[0][1] != Nil)
    end_begin(block_array[i].ptr[0][1],k);/*if id.end is begin trigger for any node*/
end algo

begin_begin(ptr, k)
/*Procedure begin_begin is called when k.begin is an begin trigger for some nodes
and correspondingly updations are done. Here k in formal parameter list is the id of
the source of synchronization. And ptr is link to the l_list of all objects for whom
k.beg is a begin enabler.*/
/*update start times of all those objects for which k.beg is begin trigger*/
while(ptr != Nil)
{
    find index i of table corresponding to ptr->id
    table[i].start_time = table[k].start_time;/*start time set to same as source object*/
    ptr = ptr->next;
}
end algo
end_begin(ptr, k)
/* Procedure end_begin is called when k.end is an begin enabler for some nodes and correspondingly updations are done for start time of destination object. ptr is link to the l_list of all objects for whom k.end is a begin enabler. Here k in formal parameter list is the id of the source of synchronization.*/
/*update end times of all those objects for which k.beg is begin trigger*/
while(ptr != Nil)
{
   find index i of table corresponding to ptr->id
   table[i].start_time = table[k].end_time;
   ptr = ptr->next;
}
end algo

algo end_end(ptr, k)
/* Procedure end_end is called when k.end is an end enabler for some nodes and correspondingly updations are done. ptr is link to the l_list of all objects for whom k.end is a end enabler.*/
while(ptr != Nil)
{
   find index i of table corresponding to ptr->id
   table[i].end_time = table[k].end_time;
   ptr = ptr->next;
}
end algo
begin_end (ptr, k)

/* Procedure begin_end is called when k.begin is an end enabler for some nodes and correspondingly end-time of destination object is set. Here k in formal parameter list is the id of the source of synchronization. ptr is link to the l_list of all objects for whom k.beg is a end enabler*/

while(ptr != Nil)
{
    find index i of table corresponding to ptr->id
    table[i].end_time = table[k].start_time;
    ptr = ptr->next;
}

end algo