A Proposal for Implementing the Concurrent Mechanisms of Ada

Xiaolin Zang

Abstract--This paper proposes a scheme for implementing the communication and synchronization mechanisms of Ada. A minimum operating system kernel is assumed first. Then the primitives and data structures used to interpret the concurrent activities are described. Ada concurrent statements are translated into the calls to certain primitives. By properly explaining some details in it, the proposal can be implemented on various computer systems and supporting environments.

Introduction

Ada provides entry call, accept statement and select statement to support the synchronized intertask message communication. The proposal for implementing these mechanisms is also based on the message communication. Our attempt is to make the proposal as independent as possible on the particular computer system and on the special support by operating system. By properly explaining some details in the scheme, it can be implemented with various buffered or non-buffered message mechanisms on singal processor, multi-processor or distributed systems.

In this paper, we will first assume a minimum operating system kernel which supports the most basic activities of tasks. Then we write out in Ada the main data structures and some primitives which interpret the communication and synchronization activities. The concurrent statements are translated into sequential statements and primitive calls.

The support by operating system kernel

We assume that the target operating system provides the following three kernel routines at least.

```
SUSPEND;                    --suspend the caller
ACTIVATE(T:TASK_NAME); --resume task T to be runnable
ALARM(ALARM_AT: TIME; EXPIRED: out BOOLEAN);
 --suspend the caller until some other task resumes the caller or the
 --time ALARM_AT is due.  EXPIRED indicates whether the time is due or
 --the caller is resumed by other task.
```

Main data structures

The compiler establishes a rum-time package for each task in an Ada source program, which maintains the data structures used in task activation, termination and communication. We consider only those which are related to entries.

An entry descriptor is set up for each entry in a task, which is of the following type

```
type ENTRY_DESCRIPTOR is
  record
    ACC_WAITING: BOOLEAN:= FALSE; --accepting task is waiting for
                                  --entry call
    SUCCESSFUL: BOOLEAN:= FALSE;  --successful rendezvous
    EMPTY: BOOLEAN:= TRUE;        --empty entry queue
    QUEUE: QUEUE_POINTER:= null;  --entry queue
  end record;
```

Type QUEUE_POINTER describes entry queue. An object of type QUEUE_ELEM is dynamically generated when an entry call is issued.

```
type QUEUE_POINTER is access QUEUE_ELEM;
type QUEUE_ELEM is
  record
    NEXT_ELEM: QUEUE_POINTER:= null;
    ELEM: CALLER_DESCRIPTOR;
  end record;
```

Type CALLER_DESCRIPTOR is defined as following.

```
type CALL_KIND is (NORMAL, CONDITIONAL, TIMED);
type CALLER_DESCRIPTOR(KIND: CALL_KIND) is
  record
    CALLER: TASK_NAME; --name of calling task
    LOCATION: MESSAGE; --location of parameters of entry call
    case KIND is
      when TIMED  => ALARM_AT: TIME; --duration of timed entry call
      when others => null;
    end case;
  end record;
```

The location of actual parameters is closely related to the particular operating system environment.

Entry call

The interpretation of entry call, accept and select statements is preformed by four primitives. The execution of the primitives must be protected. At least, primitive executing about the same entry should be mutually exclusive.

An entry call is translated into a call to primitive ENTRY_CALL which queues the calling information in specified entry queue and resumes the accepting task if which is waiting for the corresponding entry call. Then the calling task suspends until the rendezvous is completed or the duratioon is expired in the case of timed entry call.

```
procedure ENTRY_CALL(ACCEPTOR: TASK_NAME; --accepting task
                     EMTRY: in out ENTRY_DESCRIPTOR;
```

```
                                          --entry descriptor
                    CALLER: CALLER_DESCRIPTOR;
                                          --calling information
                    SUCCESSFUL: out BOOLEAN) is
                                          --successful rendezvous?
  TEMP: CALLER_DESCRIPTOR;
  EXPIRED: BOOLEAN;
begin
  if CALLER.KIND = CONDITIONAL and
     not (ENTRY.ACC_WAITING and ENTRY.EMPTY) then
    SUCCESSFUL:= FALSE;
    return;
  end if; --if accepting task is not waiting for the current entry
          --call, the conditional entry call fails to rendezvous.
  TEMP:= CALLER;
  if CALLER.KIND = TEMED then
    TEMP.ALARM_AT:= CLOCK() + TEMP_ALARM_AT;
  end if; --convert duration into time limit
  INSERT(ENTRY, TEMP);
   --queue calling task in entry queue and set ENTRY.EMPTY to be false
  if ENTRY.ACC_WAITING then ACTIVATE(ACCEPTOR); end if;
   --resume accepting task
    if CALLER.KIND = TIMED then
    ALARM(TEMP.ALARM_AT, EXPIRED);
     --wait for time limit or being resumed by accepting task
    if EXPIRED then --duration expired
      SUCCESSFUL:= FALSE;
      return;
    end if;
  end if;
  SUSPEND;                          --wait for termination of rendezvous
  SUCCESSFUL:= ENTRY.SUCCESSFUL; --successful?
  ENTRY.SUCCESSFUL:= FALSE;
end ENTRY_CALL;
```

ENTRY.SUCCESSFUL is set at the end of accept statement. The cause of unsuccessful rendezvous is usually that an accept alternative to rendezvous a conditional entry call is not selected in the selective wait of accepting task.

An entry call is translated into statement sequence

```
<Parameter pre-processing>
ENTRY_CALL(...);
<Parameter post-processing>
```

Parameter processing is closely related to particular implementation.

A conditional entry call is translated into

```
<Parameter pre-processing>
ENTRY_CALL(..., ..., ..., SUCCESSFUL);
if SUCCESSFUL then
  <Parameter post-proccessing>
  <Sequence of statements following entry call>
```

```
else
  <Parameter storage releasing>
  <Else part>
end if;
```

Timed entry call corresponds to the following objective sequence.

```
  if <Delay expression> > 0 then
    <Parameter pre-processing>
    ENTRY_CALL(..., ..., ..., SUCCESSFUL);
    if SUCCEESSFUL then
      <Parameter post-processing>
      <Sequence of statements following entry call>
      goto TAIL;
    end if;
    <Parameter storage releasing>
  end if;
  <Sequence of statements following delay>
<<TAIL>>
  <Subsequent statements of timed call>
```

Accept statement

An accept statement, which is not the first statement of an accept alternative, is translated into following statement sequence.

```
ACC_ENTER(...);
<Accept body>
ACC_LEAVE(...);
```

ACC_ENTER and ACC_LEAVE are a pair of primitives. ACC_ENTER removes the expired timed entry calls from the corresponding entry queue. The rendezvous begins if there is some entry call waiting in the queue. Otherwise, the accepting task suspends until a corresponding entry call occurs.

```
procedure ACC_ENTER(ENTRY: in out ENTRY_DESCRIPTOR) is
begin
  REMOVE_EXPIRED(ENTRY); --remove expired entry calls
  if ENTRY.EMPTY then
    ENTRY.ACC_WAITING:= TRUE;
    SUSPEND;                 --wait for an entry call
  end if;
  if ENTRY.QUEUE.ELEM.KIND = TIMED then
    ACTIVATE(ENTRY.QUEUE.ELEM.CALLER);
  end if; --resume the task issuing the timed entry call
  ENTRY.ACC_WAITING:= FALSE;
  FEED(ENTRY); --fetch calling parameters
end ACC_ENTER;
```

If the entry call to rendezvous is a timed entry call, the calling task which is waiting with the kernel routine ALARM should be resumed, then it will wait for the termination of rendezvous in primitive ENTRY_CALL. Were the calling task resumed after rendezvous, the calling task would think that no rendezvous happened if the duration was expired during the rendezvous by chance.

Procedure REMOVE_EXPIRED removes the expired timed entry calls (At this time, the calling tasks have been resumed.) from the entry queue and ENTRY.EMPTY is set to be true when the queue is empty.

Procedure FEED transfers the calling parameters to the accepting task stack, so that accepting task uses the parameters in accepting statement just as in a procedure.

Primitive ACC_LEAVE sends the parameters back to the calling task and resumes the calling task.

```
procedure ACC_LEAVE(ENTRY: in out ENTRY_DESCRIPTOR) is
begin
  FEEDBACK(ENTRY);
  ENTRY.SUCCESSFUL:= TRUE;
  ACTIVATE(ENTRY.QUEUE.ELEM.CALLER);
  REMOVE(ENTRY);
end ACC_LEAVE;
```

Procedure FEEDBACK performs the inverse action of FEED. This pair of procedures is closerly dependent on particular environment.

Procedure REMOVE unqueues the first term from the entry queue and ENTRY.EMPTY is set to be true when the queue becomes empty.

Selective wait

Since the discussion of task termination is beyond this paper, we only consider the selective waits which do not contain terminate alternatives. Primitive SELECTIVE performs the selective policies. The information interface of the primitive is designed as

```
procedure SELECTIVE(SEL: OPEN_ACC_ALTERS;
                    M, ELSE_ORDER, ORDER: INTEGER;
                    SHORTEST: DURATION;
                    SELECTED: out INTEGER );
```

where SEL is an array which records all the open accept altenatives and M is the number of such alternatives. Each alternative, including else part, is given an ordinal, according to its static order in the selective wait. If there are some open delay alternatives, ORDER and SHORTEST are the ordinal and duration of the delay alternative with the shortest duration respectively. ELSE_ORDER is the ordinal of else part, which is zero if there is no else part. SELECTED is the ordinal of selected alternative.

Suppose there are M open accept alternatives in a selective wait, array SEL is of type

```
type OPEN_ACC_ALTERS is array (1..M) of ACC_ALTER;
SEL: OPEN_ACC_ALTERS;
```

SEL is local to a particular seletive wait.

Type ACC_LATER describes an open accpt alternative.

```
type ACC_ALTER is
  record
    ORDER: INTEGER;            --ordinal
    ENTRY: ENTRY_DESCRIPTOR; --corresponding entry descriptor
  end record;
```

A selective wait with N alternatives is translated into following statement sequence.

```
ORDER:= 0; --initialize to record the ordinal of the shortest delay
SHORTEST:= DURATION'LAST; --to record the shortest duration
M:= 0;                    --to count all open accept alternatives
  <1st when-alternative>  --the first select alternative
  <2nd when-alternative>
      .
      .
      .
  <Nth when-alternative>
  <Else part>  --possible else part with ordinal N+1
<<H   >>
  SELECTIVE(SEL, M, N+1, ORDER, SHORTEST, I); --make selection
  case I of                 --I is the ordinal of selected alternative
    when   1 => goto L ;  --the first alternative is selected
    when   2 => goto L ;
      .
      .
      .
    when   N => goto L ;
    when N+1 => goto L   ; --else part is selected
  end case;
<<TAIL>>
  <Subsequent statements of selective wait >
```

The delay alternatives and else part should be mutually exclusive by the language manual. We assume that it has been checked.

In the case of accept alternative, <Ith when-alternative> is corresonding to following statement sequence.

```
<<H >>
  if <Ith condition> then --condition always true if no "when clause"
    M:= M+1;
    SEL(M).ORDER:= I;
    SEL(M).ENTRY:= <Descriptor of the entry>;
  end if;     --record an open accept alternative
  goto H   ; --go to next alternative
<<L >>
  <Accept body>
  ACC_LEAVE (...);
  <Subsequent statements>
```

```
  goto TAIL;
```

In the case of delay alternative,

```
<<H >>
  if <Ith condition> then
    if SHORTEST > <Delay expression> then
      SHORTEST:= <Delay expression>;
      ORDER:= I;
    end if;
  end if;
  goto H    ;
<<L >>
  <Subsequent statements>
  goto TAIL;
```

Else part is

```
<<L    >>
  <Else part>
  goto TAIL;
```

Primitive SELECTIVE selects a rendezvousable open accept alternative. If there is no rendezvousable one at present, the shortest duration delay is performed to wait one of the open accept alternatives becomes rendezvousable, or the else part is executed. If there is no open accept alternative, an open delay alternative or else part is selected.

```
procedure SELECTIVE(SEL: OPEN_ACC_ALTERS;
                    M, ELSE_ORDER, ORDER: INTEGER;
                    SHORTEST: DURATION;
                    SELECTED: out INTEGER) is
  RENDEZVOUSABLE, EXPIRED: BOOLEAN;
begin
  if M = 0 then      --no open accept alternatives
    if ORDER <> then --there are some open delay alternatives
      ALARM(CLOCK() + SHORTEST, EXPIRED);
       --delay, to elapse the shortest duration
      SELECTED:= ORDER;
       --to execute the subsequent statements after delay
    elsif ELSE_ORDER <> 0 then --there is an else part
      SELECTED:= ELSE_ORDER;   --to execute else part
    else
      raise PROGRAM_ERROR; --cause exception when neither open
                           --alternatives nor else part exists
    end if;
    return;
  end if; --process for the case of no open accept alternatives
  loop
    RENDEZVOUSABLE:= FALSE;
    for I in 1..M loop     --for M open accept alternatives
      REMOVE_EXPIRED(SEL(I).ENTRY);
                           --remove expired timed entry call
      RENDEZVOUSABLE:= RENDEZVOUSABLE or not SEL(I).ENTRY.EMPTY;
```

```
    end loop;
    if RENDEZVOUSABLE then --there are some rendezvousable ones
      ARBITRARY_SEL(SEL, M, SELECTED);
                              --select one and send its ordinal back
      for I in 1..M loop
        SEL(I).ENTRY.ACC_WAITING:= FALSE;
        if SEL(I).ORDER = SELECTED then
          FEED(SEL(I).ENTRY); --transfer parameters for the selected
        elsif not SEL(I).ENTRY.EMPTY then
          if SEL(I).ENTRY.QUEUE.ELEM.KIND = CONDITIONAL then
            ACTIVATE(SEL(I).ENTRY.QUEUE.ELEM.CALLER);
            REMOVE(SEL(I).ENTRY);
          end if; --for every unselected one, remove the conditional
                  --entry call and resume the calling task
        end if;
      end loop;
      return;
    end if; --a rendezvousable accept alternative has been selected
    if ELSE-ORDER <> 0 then --there an else part
      SELECTED:= ELSE_ORDER;
      return;
    end if;
    for I in 1..M loop
      SEL(I).ENTRY.ACC_WAITING:= TRUE;
    end loop; --set all open accept alternatives to be waiting status
    if ORDER = 0 then --neither open delay nor else part exists
      SUSPEND;
    else
      ALARM(CLOCK() + SHORTEST, EXPIRED);
       --delay, until duration is elapsed or some entry call occurs
      if ENPIRED then --duration expired
        SELECTED:= ORDER;
                         --to execute the statement sequence after delay
        for I in 1..M loop
          SEL(I).ENTRY.ACC_WAITING:= FALSE;
        end loop;
        return;
      end if; --some entry call wakes up the accepting task waiting
              --with SUSPEND or ALARM.  The rendezvousable alternative
              --will be selected in the next iteration.
    end if;
  end loop;
end SELECTIVE;
```

Procedure ARBITRARY_SEL selects one of the rendezvousable accept alternatives arbitrarily.

Summary

This paper describes some primitives to support Ada intertask communication and synchronization. The primitives are based on a minimum operating system kernel. The final implementation depends on the particular computer system and supporting environment. But the dependence is limited and is concentrated on the transfer of entrycall parameters and on the protection of the primitives.

The key data structure in our scheme is entry descriptor. The operations on it must be mutually exclusive. The problem should be carefully solved in the final implementation.

Reference

Ada Reference Manual, United States Department of Defence.