Temporal Specification of Ada Tasks

William Hankley and **James Peters**

Department of Computing & Information Sciences
Kansas State University, Manhattan KS 66506

Abstract -- This paper reports work on a language called Ada/TL for the specification of the temporal behavior of interacting Ada tasks in both concurrent and distributed systems. The language Ada/TL is an extension of the task specification declarations required by the Ada language. The extensions include (i) temporal assertions about rendezvous and other events of external interaction and (ii) non-temporal 'in' and 'out' assertions about parameters and other data items which flow between tasks. We use linear-time operators to specify the sequential behavior of individual tasks and branching-time operators to specify global properties about the interaction of tasks. Task specifications are intended to follow the style of Ada declarations and to be constructive inasmuch as they to lead directly to the design of task rendezvous behavior. The paper defines the representation of an externally viewable state of an Ada task and it defines operators to specify a task behavior as a sequence of state conditions. Specifications are illustrated for several examples of tasks using shared resources and interaction using both synchronous and asynchronous communication. Continuing work on specification of timing constraints and on analysis of global correctness of specifications is discussed.

1. Introduction

Specifying the behavior of programs with temporal logic was formulated by Burstall [11], Pnueli [38,39,40], and Manna [32]. Their work led to methods of proving properties of concurrent programs by Abadi [1], Barringer [3, 4, 7], Clarke [12,13], Emerson [15,16,17], Hailpern [20] Lamport [23, 24, 25, 26, 27, 28], Manna [33, 34, 35, 36, 37], and others. Recently, efforts have shifted to temporal specification of program modules rather than specification of global properties of programs, as in Barringer [7], Lamport [24], and Pnueli [42]. Usually temporal specifications have been used with respect to programs written in some abstract language or in CSP. Some work has been done on specification of Ada programs, for example Barringer [2, 5, 6] and Pnueli [41]. So far these temporal logic specification techniques have not been incorporated into a formal specification language for Ada tasking.

TSL (for Task Specification Language) by Helmhold [21] and Luckham [31] is tailored for description of the behavior of Ada tasks. Its notation for interaction of task events is similar in concept to temporal predicates, but TSL is not founded on temporal logic. TSL follows style conventions of Anna (for "annotation language for Ada"), an earlier specification language for Ada programs by Luckham [30].

We present here initial development of a formal specification language which is tailored to Ada tasks. It provides a notation for specifying the constraints on the local behavior of individual tasks and the global behavior of interaction between tasks in concurrent and distributed systems. This tasking language is called Ada/TL. It merges ideas from three different styles of specifications:

1) It is an extension of Ada task specifications defined in the Ada Language Reference Manual (ALRM) [14].
2) It uses the model-based style of VDM (Vienna Development Method), as in Bjorner [9], Lucas [29], and Jackson [22], to specify the result of procedures, including "accept" operations of tasks.
3) It uses a variation of the temporal operators defined in the logic system called UB introduced by Ben-Ari et al [8]. Linear time operators are used to specify the sequential behavior of individual tasks. Branching time operators are used to specify properties of task interaction that are not constrained by individual task behaviors.

The contribution of Ada/TL is in the merging of the model-based and temporal based specifications into an Ada framework.

The further purpose of Ada/TL is to be a design language for Ada tasking systems. To that end, it is intended that Ada/TL specifications should be constructive in the sense that task bodies can be developed from specifications. Task specifications should be concise yet easily read by Ada programmers, and easily mapped into Ada code. The requirement for constructive specifications is similar in focus to the work of Wolper [43, 44] and Manna [35] using specifications to synthesize concurrent programs. However, we do not deal with synthesis of programs in this paper.

2. ADA/TL Language

We introduce Ada/TL specifications using a simple example of tasks accessing a common buffer task. The example system is represented in Figure 1 in the graphic notation presented by Buhr [10]. Producer tasks put items in a bounded Queue task and Consumer tasks remove items from the Queue. The following sections focus separately on the ALRM framework, model-based procedure specifications, and temporal specifications.

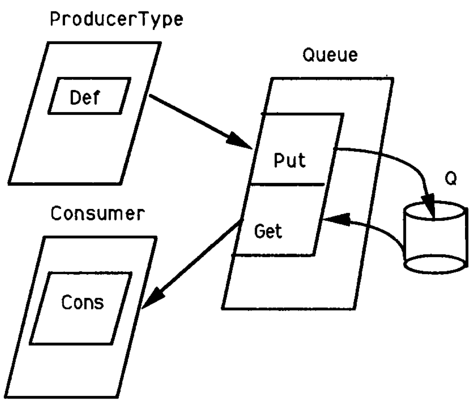

Fig. 1. Example System

2.1 Relation to Ada Specifications

The syntax rules for Ada/TL are given in the Appendix. The rules extend the general rules for declaration of Ada modules given in the ALRM but with some additions that will be explained. Specification of the example system is given in Figure 2.

```
generic
  type ItemType is pending;      -- specification parameters
  MaxSize: natural pending;

package Producers_Consumer is

  task type ProducerType is              -- task specification
    procedure Def(Item: out ItemType);  -- define Item
    property □ ◇ seq(Def(Item),  Queue.Put(Item) );
  end ProducerType;

  task Consumer is                       -- task specification
    procedure Cons(Item: ItemType);  -- consume Item
    property □ ◇ seq(Queue.Get(Item),  Cons(Item));
  end Consumer;

  task Queue is                          -- task specification
    Q : seq of ItemType;                 -- state variable
    init Q'length = 0;                   -- initial assertion
    entry Put( Item: ItemType);
      in Q'out = Q'in + Item;
    entry Get( Item: OUT ItemType);
      out Q'in = Item + Q'out;
    property Q'length ≤ MaxSize;         -- global assertion
  end Queue;

  SysProperty ∀□ (all P:  ProducerType:
      P:Def(Item)  imp ◇ Consumer:Cons(Item)   );

end Producers_Consumer;
```

Fig. 2. Sample specification.

The specification is a generic package, which means that some parameters of the system are left unspecified; namely, ItemType is identified only as pending and ProducerType is a task type, so that the number of producers is never identified. The primitive type "pending" follows from the language Gypsy in Good [18, 19]. "pending" is used to denote generic parameters which are identified but otherwise not specified. The package specification consists of three task specifications (described below) followed by a SysProperty. The SysProperty specifies temporal properties of task interactions which are not directly specified by the individual task specifications.

Task specifications include entry declarations as in the ALRM. Entry specifications may be decorated with non-temporal **in** and **out** assertions, as in VDM specifications [22]. Task specifications also define static variables and operations (procedures, functions) which are referenced as part of the task specification. We refer to these items as externally observable, in that they are used in description of the task specification but they are not accessible by other tasks. For example, the variable Q must be identified as part of the specification, but it is not intended to be accessed by tasks other than task Queue. Observable items serve in task specifications in a way analogous to that of private items in package declarations as defined by the ALRM. The last component of each task specification is the "property" assertion which specifies the temporal behavior of the subject task, including interaction with other tasks and access to either global items or its own observable items.

In summary, the following specification components are additions to Ada specifications defined in the ALRM and were at least partially illustrated in the sample specification:

+ "pending" type
+ abstract types, as sets, sequences
+ declaration of variables and operations for tasks
+ **init** and **property** assertions for packages and tasks
+ **In** and **out** assertions for procedures and entries
+ **SysProperty** assertion about task interactions

2.2 Relationship to VDM

The Vienna Development Method allows specification of the effect of operations by modeling data objects in terms of certain abstract types (sets, maps, etc). The effect of operations is defined by pre-assertions which specify constraints on parameters, by post-assertions which relate pre and post values of both parameters and module variables, and by invariant assertions about module parameters.

Ada/TL incorporates the VDM model-based style to specify the effect of individual entry operations of tasks, but with syntax tailored for Ada specifications. It uses **in** and **out** assertions for operations and **inv** (for invariant) and **init** (for initial) assertions for static data objects. Static data objects can be represented by either Ada structures or the abstract types set, map, or sequence. In the example, the variable Q is a static object of the task Queue. Q is modeled as a bounded sequence, with initial size zero. This is, in the specification Q is declared to be of the abstract type sequence, but in the task code Q must be implemented as some concrete data structure. In the procedure **out** assertions, Q'in and Q'out represent pre and post values of Q and the + operation represents concatenation of an item to the sequence. The assertions together constrain the task Queue as a bounded queue. Further details of the model-based part of Ada/TL are not covered here since that is not the focus of this paper.

2.3 Temporal Specifications

The specification of interaction of tasks is given in task property assertions and in the SysProperty assertion. Each property assertion is a linear-time temporal predicate that defines required behaviors of the subject task; the SysProperty assertion is a predicate in branching time logic that further constrains the behaviors of tasks. Temporal predicates are composed using temporal operators applied to non-temporal and control predicates. Non-temporal predicates express constraints about parameters and observable variables. Control predicates express constraints about events of execution of entry procedures and observable operations. These concepts are defined in the subsections that follow.

In the example, the "□ ◇ ,seq" operators express that the task must infinitely repeat a sequence of conditions. Each ProducerType task must repeatedly call its Def procedure and pass the Item value to the Queue.Put entry. Similarly, the Consumer task must repeatedly receive an Item value from the Queue.Get entry and pass the value to its Cons procedure. Those specifications are constructive in that they imply the code structure for the task bodies. The SysProperty requires that for all execution paths, whenever an Item value is determined by a Def operation with some task P , then eventually that value will be passed to the Cons operation within the Consumer task. In this case, the task properties are sufficiently strong that the SysProperty will be satisfied.

2.3.1 Task States

Predicates about tasks are expressed in terms of the state of a task. The state of a task is a symbolic representation of all information maintained in during execution of the task. Such execution information is maintained in the activation record of tasks and in the run-time kernel that controls task interaction. The symbolic information is not actually computed by each task, but the symbolic representation is needed to order to express the specifications. Not all of the components of the state are defined within the Ada language, but they are part of the underlying semantic model of Ada and they are needed to write specifications. For purposes of just writing specifications of tasks, no formal semantics for Ada or Ada/TL are given. Such formal semantics would be defined in terms of the task state. The state components are described informally below.

(1) All observable variables of the task (declared in the task specification) are part of its state. In the task Queue example, Q is an observable variable.

(2) Some components of the state are represented as fields of a new task attribute called STATE (even though it is only part of the full state). STATE has type:

```
abs type
   TaskStateType is record
   TaskName : TaskNameType;        --identifies task
   StateName: StateNameType;       --accepting, waiting
   CallStack : seq of CallRec;     --stack of call records
   Op          : NameType   --Entry, Proc, or Exception called
   Time        : time;             --time of start of current Op
   CallerId    : TaskNameType;     --rendezvous client
   Timed       : Boolean;          --true for timed entry
   DeltaTime   : time ;            --remaining time for entry
end TaskStateType;
```

The declaration "abs type" indicates the type is composed of some abstract structures (the sequence). StateNameType is an enumeration of state names drawn in part from Burns [45].

```
StateNameType = (calling, pending_accept, accept,
                 completing_accept, proc, raise, ...);
```

with the following meanings:

StateNameType	Explanation
calling	Task is a client calling an entry in some server task.
pending_accept	Task suspended prior to "arrival" of a client task.
accept	Server task executing an accept statement.
completing_accept	Server task transferring any OUT parameters of accept statement.
proc	Task either elaborating or executing a procedure.
raise	Task raising an exception.

The CallStack field represents the execution stack of procedure activation records. The Op field identifies the name of the most recent entry accepted, or procedure or entry called, or exception raised. The Time field contains the time of the start of the current Op measured with the appropriate local clock. During a rendezvous, the CallerId is the name of the client task. If an "accept" or entry call is timed, then the Timed field is true and DeltaTime is the remaining time for the delay operation.

Within a task specification we allow the STATE components to be referenced as STATE.field , without explicitly writing task_name'STATE.field. In referring to other tasks or in writing a SysProperty, the task name must be written, but word STATE can be omitted. For example, the expression T'field refers to the "field" component of the STATE of task T.

(3) Associated with each entry procedure is the new attribute QUEUE of type EntryQueue, which is a sequence of EntryRec's. If E is an entry, then E'QUEUE has length E'COUNT.

(4) Finally, the task state also includes the parameters of each active procedure call and accepted entry. Parameters can be referenced in IN and OUT assertions, as in the assertions for the entry Put in the Queue example. Also, parameters can be referenced in temporal "property" and SysProperty assertions. Interpretation of parameters in temporal predicates is covered in the next section.

2.3.2 Control Predicates

Control predicates express conditions about the point of control of task execution. Control predicates are composed using operators "at" and "in" defined below, but the "at" operator may be omitted. "at" is the default if it is omitted. Points of execution can only be expressed relative to structures visible in the specification (procedures, entries, and exceptions).

Control predicate for task T	Meaning
at(*EntryName*)	T'StateName = pending_accept and T'Op = *EntryName*
at(*Task.EntryName*)	T'StateName = calling and T'Op = *Task.EntryName*
at(*ProcName*)	T'StateName = proc and T'Op = *ProcName*
at(*ExceptionName*)	T'StateName = raise and T'Op = *ExceptionName*
in(*EntryName*)	at (*EntryName*) or (T'StateName = accept and T'Op = *EntryName*)
in(*ProcName*)	at (*ProcName*) or *ProcName* in T'CallStack

Although not shown above, control predicates allow parameters for procedures and entries. "in" parameters are required to have a value already bound in the surrounding context. "out" parameters interpreted as becoming bound to some value determined by the procedure or entry any satisfying all associated assertions. (This follows the concept of unifying of parameters in logic languages such as Prolog.) In the example SysProperty, the Item is bound in the Def procedure and the same bound value satisfies the Cons procedure.

2.3.3 Temporal Predicates

Temporal predicates are composed of non-temporal predicates (operators and quantifiers of first order predicate calculus), control predicates, and temporal operators. The temporal operators are defined in this section. Whereas non-temporal predicates are interpreted for a single state, temporal predicates are interpreted over a sequence of states, which is called a path of execution. As indicated before, we use linear-time predicates for paths of execution of a single task and we use branching-time predicates for paths of execution of a system of interacting tasks.

Let path $h = (s_0, s_1, s_2, \ldots)$ represent the sequence of observable states of a task T. Each state s_i is a tuple of the state components identified in Section 2.3.1. Let P, Q, P_1, ... be predicates over states of T, and let i be a natural number which denotes the index of a "current" state. The notation $K_{i,h}(P)$, called a Kripke structure, denotes that P holds (is true) in the i th state of h. The primary linear time operators are defined below. Note that the Kripke structure is not actually written in specifications; it is merely used to define the operators. Note also that the path of execution of the Task T is not actually known; it is merely used in symbolic form to explain "property" assertion. The purpose of using a predicate P in a constructive specification is to be able to design the task body so that any execution path of T will satisfy the specification.

Name	Meaning
always	$K_{i,h}(\Box P) == (\text{all } j\text{: } j \geq i\text{: } K_{j,h}(P))$
eventually	$K_{i,h}(\Diamond P) == (\text{exists } j\text{: } j \geq i\text{: } K_{j,h}(P))$
nexttime	$K_{i,h}(\bigcirc P) == K_{i+1,h}(P)$
before	$K_{i,h}(P \text{ before } Q) ==$ $(\text{exists } j,k\text{: } i \leq j < k\text{: } K_{j,h}(P) \text{ and } K_{k,h}(Q))$
sequence	$K_{i,h}(seq(P)) == K_{i,h}(\Diamond P)$ $K_{i,h}(seq(P_1, P_2, \ldots, P_n)) ==$ $K_{i,h}(P_1 \text{ before } seq(P_2, \ldots, P_n))$
infinitely often	$\Box \Diamond P$

Branching time operators are defined in a similar fashion, but over a composite path of all tasks of a system. A system consists of global variables and a set of interacting tasks. Let a system state S_i represent a tuple of the global variables of the system and the states of all tasks of a system. Let $H = (S_0, S_1, S_2, \ldots)$ represent a sequence of states of a system. Each component of H (except the global variables component) is a path h of one of the tasks of the system. Even though each task of a system may be deterministic, the total system behavior is generally nondeterministic. So, even though the path of each task satisfies its own property (that is, each task behaves in the proper sequence of states), there may be many possible interleavings to the tasks, and hence many possible paths H. (That is why the SysPropery assertion is necessary.) The Kripke structure for system paths is interpreted as follows. Let $\mathbb{H}$ be the set of system paths that contain a current state S_i. Then,

$K_{i,\mathbb{H}}(P) == (\text{all } H \text{ in } \mathbb{H} : K_{i,H}(P))$. Branching time temporal operators $\forall\Box$, $\forall\Diamond$, $\forall\bigcirc$ are all defined in the following form, where # represents anv of the linear time operators :

$$K_{i,\mathbb{H}}(\forall \# P) == (\text{all } H \text{ in } \mathbb{H} : K_{i,H}(\# P))$$

3. SMALL TASKING SYSTEM SPECIFICATIONS

This section presents three examples of tasking specifications together with some skeletal task bodies. The objective is to demonstrate the specification style of the Ada/TL language and to suggest its constructive relation to program design. Tasks in these synchronization examples are grouped together in packages with corresponding system properties. The three examples belong to a family of tasking systems. Each member of this family consists of a collection of tasks competing for access to a shared resource. A Server task provides the necessary synchronization so that no more than one task at a time has access to the shared resource. In each of these examples, we first show a tasking system in the graphical style suggested by Buhr [10]. This is followed by an Ada/TL specification and the corresponding task bodies.

3.1 *Synchronized Access to a Shared Resource*

Figure 3 shows a configuration, called Synchronizer_1, of a collection of tasks which compete for access to a shared resource through a Server. The Server performs a desired operation on the resource. The Ada/TL specification for Synchronizer_1 is given in Figure 4.

In this specification, the Request entry of the Server calls the Perform procedure to compute OpChoice (the desired operation) on the shared resource. The "map" predicate is a primitive relation. The "out" assertion of Perform indicates that Result is determined as a function (mapping) of OpChoice and "resource".

The specification of User tasks states that each such task repeatedly performs the following sequence actions to use the shared resource:

Define OpChoice
Rendezvous with Server
Consume Result

Finally, no system property is required for Synchronizer_1. Mutually exclusive use of the resource results from the Ada rendezvous. The Server hold the resource and the Server can only service one request at a time .

The tasking system specification can be implemented with the following task bodies:

```
task body Server is
  with resource; use resource;
  procedure Perform(OpChoice: IN OpType;
                    Result   : OUT ResultType);
   --code to perform operation OpChoice on resource
    -- and return result
begin
   loop
   accept  Request(OpChoice: IN OpChoice;
                   Result    : OUT ResultType) do
        Perform( OpChoice,  Result );
     end Request;
   end loop
 end Server;
```

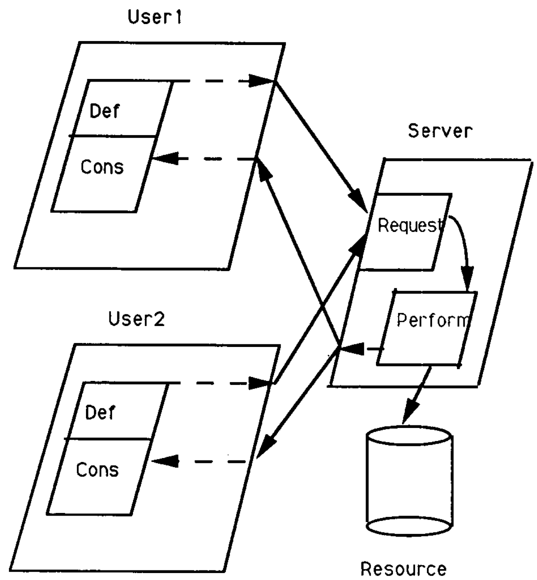

Fig. 3. Synchronizer_1

```
generic
 type ResourceType is pending;
 type ResultType is pending;
 type OpType is pending;

package Synchronizer_1 is

 task Server is
  resource: ResourceType;
  procedure Perform( OpChoice: IN OpType;
                     Result  : OUT ResultType);
     --perform operation OpChoice on resource and
      -- return Result
    out  map(OpChoice, resource, Result);
  entry Request(OpChoice: IN OpType;
                Result    : OUT ResultType);
     --process request from user to perform operation
    out  Perform(OpChoice,Result);
  property □ ◇  Request;
 end Server;

 task type UserType is
   procedure Def(OpChoice: OUT OpType);
    --select operation to be performed on resource
   procedure Cons(Value: IN ResultType);
    --consume Value
   property
      □ ◇  seq(Def(OpChoice),
               Server.Request(OpChoice, Result),
               Cons(Result)  );
 end UserType;

 SysProperty   true;
   --mutual exclusion ensured by Server rendezvous

end Synchronizer_1;
```

Fig. 4 Synchronizer_1 Specification

```
task body UserType is
  procedure Def(OpChoice: OUT OpType);
   --code to select operation OpChoice to be performed on
   --resource
  procedure Cons(Value: IN ResultType);
   --code to consume value
begin
   loop
      Def(OpChoice);
      Server.Request(OpChoice, Result);
      Cons(Result);
   end loop;
end UserType;
```

The purpose of showing the structure of the task bodies is to point out that the body structure follows directly from the looping and sequence structure of the specification. We believe that Ada/TL task specifications can be used to generate the structure of the corresponding task bodies.

3.2 Capability Passing

A variation of the tasking system in Section 3.1 is one in which each of the competing tasks acquires the capability to access a shared resource. A graphical rendition of this new configuration, called Synchronizer_2, is system is shown in Figure 5. The specification of Synchronizer_2 is shown in Figure 6.

In Synchronizer_2, the Server task regulates access to the shared resource by first granting access capability to a client task and then later retracting access rights to the same client. The Server property restricts a rendezvous between a client task and the Server Done entry with

TaskToGetCapability = TaskToReleaseCapability

A client User task repeatedly performs the following sequence of actions:

Define the OpChoice to be performed
Get access rights from the Server
Perform the desired OpChoice on the resource
Release access rights to the Server
Consume the result obtained from the resource.

The SysProperty for Synchronizer_2 is a safety property. That is, enforcement of this SysProperty guarantees that nothing "bad" happens relative to the shared resource. We want to guarantee that no more than one User task accesses the shared resource at the same time (mutual exclusion property). This system property is expressed in branching time and says that for any two User tasks U1 and U2, they are not both "in" (executing) the Perform operation. This mutual exclusion property is achieved by the tasking system protocol, but that is not proved by the specification. Such proof is left for further work.

We leave it as an exercise that implementation of the task bodies will have the same structural form as the specification.

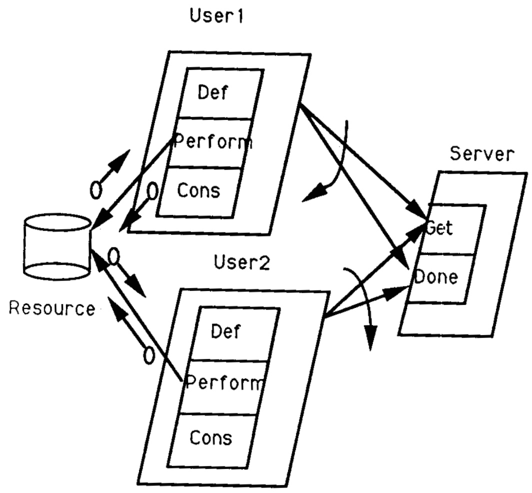

Fig 5. Synchronizer_2

```
generic
  type ResourceType is pending;
  type ResultType is pending;
  type OpType is pending;

package Synchronizer_2 is

  resource: ResourceType;

  task Server is
   entry Get(TaskToGetCapability:          IN TaskName);
   entry Done(TaskToReleaseCapability:     IN TaskName);
   property
    □ ◇ seq( Get(TaskToGetCapability),
              (Done(TaskToReleaseCapability)
                and TaskToGetCapability =
                    TaskToReleaseCapability) );
  end Server;

  task type UserType is
   procedure Def(OpChoice: OUT OpType);
      --select operation to be performed on resource
   procedure Perform( OpChoice: IN OpType;
                      Result   : OUT ResultType);
      --perform operation OpChoice on resource and
      -- return Result
     out  map(OpChoice, resource, Result);
   procedure Cons(Value: IN ResultType);
      --consume Value
   property
     □ ◇ seq(Def(OpChoice),
              Server.Get(STATE.TaskName),
              Perform(OpChoice, Result),
              Server.Done(STATE.TaskName),
              Cons(Result) );
  end UserType;
   SysProperty
     ∀□ (all U1, U2: UserType and U1 /= U2:
          not(in U1: Perform and in U2: Perform));

 end Synchronizer_2;
```

Fig.6 Synchronizer_2 Specification

3.3 Time Entry calls

This section introduces a third version of the tasking system with a simple timed entry call. Each User task requests service from the Server task and then waits for a response subject to a time limit on how long it takes the Server to respond. If the response is late, an alarm is raised. There is no recovery mechanism for this simple example. A graphical overview of this new tasking system, called Synchronizer_3, is given in Figure 7. The specification is given in Figure 8.

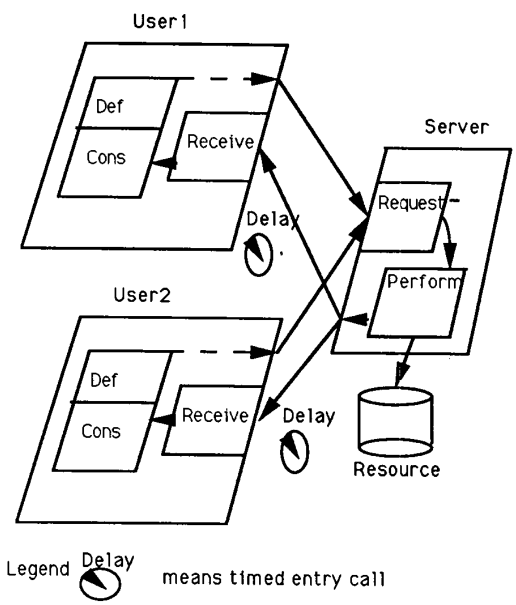

Figure 7 Synchronizer_3

```
generic
 type ResourceType is pending;
 type ResultType is pending;
 type OpType is pending;

package Synchronizer_3 is

 task Server is
  procedure Perform( OpChoice: IN OpType;
                     Result  : OUT ResultType);
     --perform operation OpChoice on resource and
     -- return Result
    out map(OpChoice, resource, Result);
  entry Request(OpChoice: IN OpType);
     --receive request from user to perform operation
  property
   □ ◇ seq((Request(OpChoice)
               and Who = STATE.CallerId),
              Perform(OpChoice, Result),
             Who.Receive(Result) );
 end Server;

 task type UserType is
  No_Response: exception;
  procedure Def(OpChoice: OUT OpType);
     --select operation to be performed on resource
     --determine an Id for the request
  procedure Cons(Result: IN ResultType);
     --consume Result returned by Server
  entry Receive(Result: IN ResultType);
     --receive Result of OpChoice from Server
  property
   □ ◇ seq(Def(OpChoice),
            Server.Request(OpChoice),
            ( (Receive(Result) and STATE.DeltaTime<0.2)
              or
              (STATE.DeltaTime≥ 0.2 and No_Response)),
             Cons(Result) );
 end UserType;

end Synchronizer_3;
```

Fig. 8 Synchronizer_3 Specification

In the Synchronizer_3 specification, the Server obtains an OpChoice from a client via the Request entry. After the Request rendezvous, the Server does the Perform and then attempts to send this Result back by calling the Receive entry in the client task. The client puts a time limit on how long it waits before it gets a response from the Server. If the time limit for the response is exceeded, the User task is "at raise No_Response" which is the end of the execution path for the task. There is no specification of recovery behavior.

A possible implementation structure of task bodies for Synchronizer_3 is shown below. The implementation assumes User tasks are contained in a family called UserFamily. An additional entry Init is provided to set the Index for each user.

```
task body Server is
  with resource; use resource;
  Result: ResultType;
  Who:    User_Index;
  procedure Perform(OpChoice: IN OpType;
                    Result  : OUT ResultType);
   --perform operation OpChoice and return Result
   begin
   loop
     accept Request(OpChoice: IN OpType;
                    Who: In User_Index) do
     end Request;
       Perform(OpChoice, Value);   --access to resource
         UserFamily(Who).Receive(Value); --return  Value
   end loop;
 end Server;
```

```
task body UserType is
  Op: OpType;
  Index: User_Index;
  Result: ResultType;
  procedure Def(OpChoice: OUT OpType);
   --Select operation to be performed on resource
  procedure Cons(Value: IN ResultType);
  --Consume Value
begin
  accept   Init(Index: IN natural);
   loop
     Def(Op);
     Server. Request(Op, Index);
    select
      accept Receive(Result:  IN ResultType);
      or
       delay 0.2;    --wait 2 10ths sec before alarm
      raise No_Response;
    end select;
    Cons(Result);
  end loop;
end UserType;
```

4. Conclusion

This paper has presented initial concepts of the new specification language ADA/TL. The language integrates concepts of Ada specifications, temporal predicates, and VDM. We feel that this style of specification language can be used by ADA developers and it can have the mathematical foundations of temporal logic and VDM. ADA/TL specifications have both a logical style and "pseudo-code" style of a design language. This conforms to the guidance given by Lamport [23] for simple specification of concurrent systems. We conjecture that ADA/TL specifications are constructive in the sense that they lead to the structure of the target system code. This has been illustrated by the examples and it is one subject of continuing work. Development of ADA/TL specifications should be supported by development tools such as a syntax directed editor and a structure checker (parser and semantic checks). An important part of validating specifications is to verify that task properties are consistent with each other and with the system property. Towards that end, we are developing a proof system for ADA/TL specifications [46].

References

[1] M. Abadi, "Temporal Logic Theorem Proving", Ph.D. Dissertation, Stanford Univ., 1987, 160pp.

[2] H. Barringer, "Axioms and Proof Rules for Ada Tasks", *IEE Proceedings*, Part E: Computers and Digital Techniques, vol. 129, No. 2 (Mar 1982), 39-48.

[3] H. Barringer, R. Kuiper, "A Temporal Logic Specification Method Supporting Hierarchical Development", Tech Report, Univ of Manchester (Nov. 1983).

[4] H. Barringer, R. Kuiper, A. Pnueli, "Now You May Compose Temporal Logic Specifications", *Proceedings of the 16th ACM Symposium on Theory of Computing*, Washington, D.C. (1984), pp. 51-63.

[5] H. Barringer, "Specifying Ada Tasks: Where we are and where we need to be", *Ada UK News*, vol. 6, part 2 (Ap 1985), 24-31.

[6] H. Barringer, "A Proof System for Ada Tasks", *The Computer Journal*, vol. 29, No. 5 (Oct. 1986), 404-415.

[7] H. Barringer, "Using Temporal Logic in the Compositional Specification of Concurrent Systems", University of Manchester TR UMCS-86-10-1, Oct. 29, 1986.

[8] M. Ben-Ari, Z. Manna, A. Pnueli, "The Temporal Logic of Branching Time", *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*. Jan. 26-28, 1981, 164-175.

[9] D. Bjorner, C. Jones, *Formal Specification and Software Development*. NJ: Prentice-Hall, 1982.

[10] R. Buhr, *System Design with Ada*, NJ: Prentice-Hall, 1984.

[11] M. Burstall, "Program proving as hand simulation with a little induction", *Proceedings of IFIP Congress*, 1974, Stockholm. Amsterdam: North-Holland, pp. 308-312.

[12] E. Clarke, M. Browne, E. Emerson, A. Sistla, "Using Temporal Logic for Automating Verification of Finite State Systems" in *Logics and Models of Computer Science*. NY: Springer-Verlag, 1985, 3-26.

[13] E. Clarke, O. Grumberg, "Research on Automatic Verification of Finite-State Concurrent Systems", Carnegie Mellon TR CMU-CS-87-105, Jan. 1987.

[14] US Dept Defense, *Reference Manual for Ada Programming Language*, ANSI/MIL STD 1815A-1983. NY: Springer-Verlag, 1983.

[15] E. Emerson, J. Hailpern, "Decision Procedures and Expressiveness in the Temporal Logic of Branching Time", *14th Annual ACM Symposium on the Theory of Computing*, 1982.

[16] E. Emerson, "Alternative Semantics for Temporal Logic", *Theoretical Computer Science*, vol. 26 (1983), 121-130.

[17] E. Emerson, T. Sadler, J. Srinivasan, "Efficient Temporal Reasoning" (Extended Abstract). *Proceedings of 16th Annual ACM Symposium on Principles of Programming Languages*, Austin, TX, Jan. 1989, 166-176.

[18] D. Good, et al. "Report on the Language Gypsy: Version 2.0", TR ICSCA-CMP-10, Institute for Computing Science, UT at Austin, 1978

[19] D. Good, "Revised Report on Gypsy 2.1", Tech Report, Institute for Computing Science, UT at Austin, July, 1985.

[20] B. Hailpern, "Verifying Concurrent Processes Using Temporal Logic", *Lecture Notes in Computer Science*, 129. Springer-Verlag: NY, 1981.

[21] D. Helmhold, D. Luckham, "TSL: Task Sequencing Language", *Proceedings of the 1985 SIGAda International Conference*, pp. 255-274.

[22] M. Jackson, "Developing Ada programs using the Vienna Development Method", *Software--Practice and Experience*, vol. 15, no. 3, March 1985, pp. 305-318.

[23] L. Lamport, "Specifying Concurrent Program Modules", *ACM Trans. on Programming Languages and Systems*, vol. 5, no. 2, April 1983, 190-222.

[24] L. Lamport, F. Schneider, "Formal Foundation for Specification and Verification", *Lecture Notes in Computer Science 190*, pp. 203-286. Springer-Verlag: New York, 1982.

[25] L. Lamport, "Reasoning about Nonatomic Operations", *Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages*, Austin, TX, (1983), 28-37.

[26] L. Lamport, "An Axiomatic Semantics of Concurrent Programming Languages" in *Logics and Models of Concurrent Programs* edited by K.R. Apt. Series F: Computer and System Sciences, Vol. 13. Springer-Verlag, Berlin, 1985, pp. 77-122.

[27] L. Lamport, "Control Variables are Better than Dummy Variables for Reasoning about Program Control", Tech Report, Digital Equip. Corp., May 5, 1986.

[28] L. Lamport, "A Simple Approach to Specifying Concurrent Systems", *CACM*, vol. 31, no. 1 (Jan. 1989), 32-47.

[29] P. Lucas, "Main Approaches to Formal Specification" In *Formal Specification & Software Development.*, D. Bjorner, C. Jones(eds.), NJ: Prentice-Hall, 1982.

[30] D. Luckham, F. Von Henke, "An Overview of Anna, A Specification Language for Ada" *IEEE Software,* Mar. 1985, 9-22.

[31] D. Luckham, et al. "Task Sequencing Language for Specifying Distributed Ada Systems TSL-1", Technical Report No. CSL-TR-87-334, July 1987. Computer Systems Laboratory: Stanford University, Stanford, CA 94305-2192, 42 pp.

[32] Z. Manna, A. Pnueli, *The Modal Logic of Programs, Automata, Languages and Programming*, LNCS 79. Berlin: Springer-Verlag, 1979, 385-409.

[33] Z. Manna, A. Pnueli, "Temporal Verification of Concurrent Programs: The Temporal Framework of Concurrent Programs", *The Correctness Problem in Computer Science* R. Boyer, J Moore. (eds), Orlando, FL: Academic Press, 1981, 215-272.

[34] Z. Manna, A. Pnueli, *Verification of Concurrent Programs: Temporal Logic Principles*, LNCS 131. NY: Springer-Verlag, 1981.

[35] Z. Manna, P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications", *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 1, Jan. 1984, pp. 68-93.

[36] Z. Manna, A. Pnueli, "How to Cook a temporal proof system for your pet language", *Proc. ACM Symposium on Programming Languages*, Austin, TX, (Jan. 1983), 141-154.

[37] Z. Manna, A. Pnueli, "Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs", *Science of Computer Programming*, 4, 3 (1984), pp. 257-290.

[38] A. Pnueli, "The Temporal Logic of Programs", *18th Annual Symposium on the Foundations of Computer Science*, IEEE, Nov. 1977, pp. 46-57.

[39] A. Pnueli, "The Temporal Semantics of Concurrent Programs" in *Lecture Notes in Computer Science 70*, Springer-Verlag, NY, 1979, pp. 1-20.

[40] A. Pnueli, "The temporal Logic of concurrent programs", *Theoretical Computer Science*, vol. 13 (1981), pp. 45-60.

[41] A. Pnueli, W. DeRoever, W.P. "Rendezvous with Ada--A Proof Theoretical View", *Proceedings of the AdaTEC Conference on Ada*, Arlington, VA, Oct. 6-8, 1982, 129-137.

[42] A. Pnueli, "In transition from global to modular temporal reasoning about programs", in *Logics and Models of Concurrent Programs* , K. Apt. (ed) Series F: Computer and System Sciences, vol. 13. Springer-Verlag.

[43] P. Wolper, "Synthesis of Communicating Processes from Temporal Logic", Ph.D. Dissertation, Stanford Univ., 1982, 111 pp.

[44] P. Wolper, "Expressing Interesting Properties of Programs in Propositional Temporal Logic", *Proceedings of 13th Annual ACM Symposium on Principles of Programming Languages*, Jan. 1986, 184-193.

[45] A. Burns, A. Lister, A. Wellings, *A Review of Ada Tasking*, LNCS 262, Springer-Verlag, 1987.

[46] W. Hankley, J. Peters, "A Proof Method for Ada/TL", TR-CS-89-11, Kansas State Univ, 1989.

APPENDIX ADA/TL GRAMMAR

This grammar gives the extensions to the grammar of Ada specifications given in the ALRM. Items in italics refer to the corresponding item as defined in the ALRM. Items on the left hand side of a production with the same name as an item in the ALRM grammar redefine the item.

Legend: [ ] indicates 0 or 1 occurrences
{ } indicates 0 or more occurrences
/* */ delimit comments for this grammar

AdaTLspec ::= *generic_specification* | *package_specification*

basic_declaration ::= *basic_declaration*
| global_assertion
| def_declaration

type_declaration ::= *type_declaration*
| abstract_type_declaration

abstract_type_declaration ::= **abs** *type_declaration*

type_definition ::= *type_definition* [**pending**]
| seq of subtype_indication
| set of subtype_indication
| **pending**

def_declaration ::= **def** *function_call* = *boolean_expression*

task_specification ::=
task [type] *identifier* **is**
{TaskParts}
end TaskName ;

TaskParts ::= *entry_declaration*
| *representation_clause*
| subprogram_specification
| StateVar
| **init** *expression*
| local_property
| *exception_declaration*

subprogram_speccification ::=
subprogram_specification
[--**in** inv_assertion]
[--**out** inv_assertion]

entry_declaration ::=
entry_declaration
[--**in** inv_assertion]
[--**out** inv_assertion]

StateVar ::= *object_declaration*
init_assertion ::= *expression* = *constant*
inv_assertion ::= expr

global_assertion ::= **SysProperty** branching_expr
/* can occur only in the system package spec */

local_property ::= **property** temporal_expr ;
/* can occur only within task specifications */

branching_expr ::= branching_time_op quant_expr

quant_exp ::=
(quant name {,name} : name_constr : quant_exp)
| temporal_expr
quant ::= all | exist
name_consts := name_const { logical_operator name_const}
name_const ::= type_mark | expression

temporal_expr ::=
[temporal_op] expr [binary_op temporal_expr]

expr ::= *expression*
| seq(temporal_expr { , temporal_expr })
| control_predicate

primary ::= *primary*
| *exception_name*
| (temporal_expr)

temporal_op ::= linear_time_op
| branching_time_op

linear_time_op ::= □ | ◇ | ○ | □ ◇
branching_time_op ::= ∀ linear_time_operator
binary_operator ::= imp
control_operator ::= at | in

control_predicate ::=
[control_operator] [*task_name* :] control_point
control_point ::= *procedure_call_statement*
| *entry_call_statement*
| *abort_statement*
| *delay_statement*
| *exception_name*

simple_name ::= *identifier* | _
/* _ is "don't care" symbol */
