SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 22(4), 317–348 (APRIL 1992)

ARTK-M2: A Kernel for Ada Tasking Requirements: an Implementation and an Automatic Generator

JORGE L. DÍAZ-HERRERA , RONALD D. GRAFT AND DOUGLAS B. RUPP
Department of Computer Science, George Mason University, 4400 University Drive, Fairfax, Virginia 22030-4444, U.S.A.

SUMMARY

A run-time kernel, ARTK-M2, supporting Ada tasking semantics is discussed; full support for task creation, synchronization, communication, scheduling, and termination is provided, together with all options of the Ada rendezvous. An implementation in Modula-2 is presented and a method for automatically translating Ada programs into semantically equivalent Modula-2 programs with corresponding kernel calls is introduced. A parser generator and an attribute grammar were used for the automatic translation. A subset of the Ada Compiler Validation Capability was processed to test the implementation and to illustrate the translation mechanism. The kernel is applicable to the study of real-time control systems; it can also serve as a baseline for studying implementation alternatives of Ada concepts, such as new scheduling algorithms, and for analysing new language constructs. Work is under way to implement some of the changes to the Ada tasking model being proposed as a result of the language revision (Ada9X). Finally, through proper extensions, ARTK-M2 can form an integral part of programming tools such as an Ada compilation system and a distributed kernel for multi-processing environments.

KEY WORDS Run-time kernels Ada tasking Modula-2 Parser generators

INTRODUCTION

Sequential and concurrent languages define a spectrum; at the high-end, a number of high-level languages provide constructs for specifying concurrent execution (e.g. Ada [1]); and at the low-end we have languages with no concurrent programming constructs but access to operating systems services used to emulate concurrency. Concurrent programming languages do require special run-time support, and as a result, whether or not processes are executed in true or apparent concurrency becomes totally transparent to the programmer. Modula-2 [2] is somewhere in the middle of such a range; it provides primitives useful for implementing concurrency but does not support a full set of high-level features. Although this has the disadvantage of requiring the programmer to provide implementation details of the underlying model of concurrency, it makes the language not only easier to implement but ideal for writing such run-time support systems or kernels.

In this paper we describe ARTK-M2. a run-time kernel, written in Modula-2 that implements Ada tasking semantics. All options of the Ada rendezvous including

0038–0644/92/040317–32$16.00

Received 26 February 1990
Revised 20 November 1991

conditional entry and accept calls, task priorities, family of entries, parameter passing, and the delay and terminate alternatives of the select statement are provided. This kernel can be used as a basis for programming concurrent applications using the Ada tasking model by placing kernel calls at the appropriate places in a Modula-2 program. The system can also be used as an initial run-time system for an Ada compilation environment. We illustrate this point here by automatically translating Ada tasking programs into their semantically equivalent Modula-2 programs using a parser generator. Finally, ARTK-M2 provides a useful workbench to investigate implementation and other issues [3–9] which can be studied both quantitatively and qualitatively. This is specially relevant in the light of Ada language revisions [10] and for the design of new languages in general.

THE ADA TASKING MODEL

A sequential Ada program consists solely of sequential actions executed as a single sequential process running on a single logical processor. Concurrent Ada programs consist of multiple sequential processes that can be executed simultaneously in the sense that each runs on its own logical processor. Each concurrent Ada process is defined by a *task program unit.* An Ada task, which is the unit of logical concurrency, proceeds independently except at points where it needs to synchronize with other tasks. Tasks may be implemented on multicomputers, multiprocessors, or with processor multiplexing on a single physical processor; furthermore, a single task may be implemented as executing in different physical processors running in parallel. An important notion is that the actual implementation approach taken by an Ada compilation system is hidden from the programmer and does not have an effect on the meaning of a correct concurrent Ada program.

Program and task structure

The notion of a textually monolithic program has disappeared from Ada. An Ada 'program' is a hierarchical collection of *library units, secondary units* and *subunits.* Program units, Ada's basic building blocks, correspond to *subprograms, packages* and *tasks;* where subprograms include both functions and procedures. All Ada program units are defined in the same structural way as consisting of two parts, namely a specification and an accompanying body. The syntax is oriented toward supporting the physical segmentation of software; separately compiled units are collectively referred to as *compilation units.* This structure is illustrated graphically using the HMD notation [11] in Figure 1. More specifically, a library unit is a separately compiled subprogram or package specification, a secondary unit is the corresponding separately compiled body of a library unit, whereas a subunit is the separately compiled body of a local program unit nested in a secondary unit or in another subunit. This scheme of separately compiled bodies yields a tree-like hierarchy. Library units are imported into a compilation unit; this defines a linear partial ordering of units, or layers of abstractions.

Items defined in the specification part of a program unit are visible, i.e. exported, outside the unit, whereas items defined in a program unit body are totally hidden and not accessible from the outside. A unit's execution semantics is defined by the statements in the *unit body.* A *unit specification* defines an interface separating the

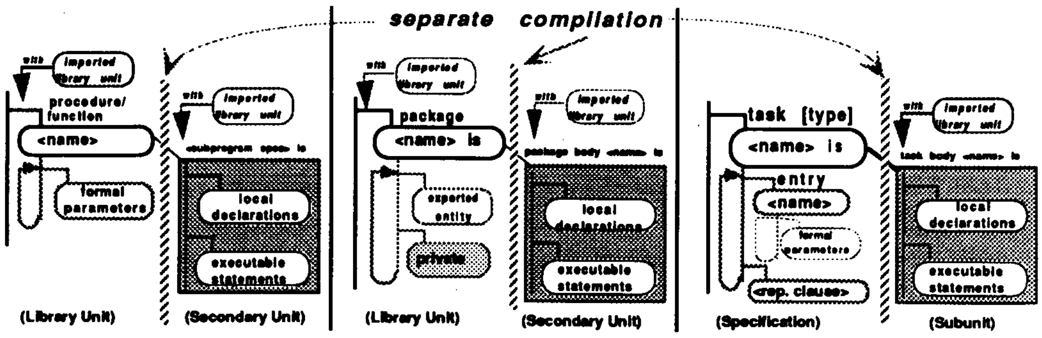

Figure 1. Ada library units and task structure

corresponding body from the rest of the software. and includes the unit name and an optional list of exported items; in addition, a package specification may include a non exported *private* section. The purpose of this private section is to provide non-exported information needed to compile the specification separate from its body and from other units that use it. A task specification defines the interface of the task with other tasks and with the 'main program'. Although the word 'program' is not part of the Ada reserved words list, the idea of a main procedure still persists; it actually takes the form of a subprogram library unit, and acts as if called by some enclosing *environment task.* Tasks cannot be library units, and thus must be declared inside another unit. If the task specification includes the word type, then it defines a task type. An object of a task type designates a task having the entries, if any, of the corresponding task type, and its body. A task specification without the word type defines a single task object of an anonymous task type; this type is declared by the compiler.

Figure 2 illustrates task declarations. In this example, the library unit Dining_Philosophers declares two (anonymous type) tasks, Forks and Chairs, and an array of Num of task type Philosopher. The corresponding proper bodies are submitted separately as three subunits as indicated by the body stubs in lines 24–26. Excerpts of these subunits are presented in Figure 3.

Task activation and termination

In Ada, task creation is done implicitly by the run-time system during the *elaboration* of a task object of the corresponding type. Elaboration refers to the run-time processing of declarations. The creation of the environment task and the 'main subprogram' is also the responsibility of the run-time system. Elaboration of a task specification establishes the corresponding task type. The *activation* of a task causes itself the elaboration of the declarative part of a task body, which may in turn contain other (local) tasks, thus forming a hierarchy of (sub) tasks. During task activation, tasks are initiated for execution after the elaboration of their declarative part is complete.

Each created task depends on a *Master* which may be another task, a currently executing block or subprogram, or a library package. Blocks behave like in-line anonymous parameterless procedures. Notice that a local package is never a Master.

```
with CALENDAR, TEXT_IO ;
use CALENDAR, TEXT_ IO;
procedure Dining_Philosophers is . . .
task Forks is
    entry Pick_Up  (Fork) ; -- a family of entries
    entry Put_Down (F : Fork) ;
end Forks;

task Chairs is
    entry Give (Me   : out aChair ) ;
    entry Here_is (My : aChair) ;
end Chairs ;

task type Philosopher is
    entry Birth (Me : Name; Life_Time : DURATION := 0.0) ;
 end Philosopher ;

task body Forks is separate;
task body Chairs is separate;
task body Philosopher is separate;
begin -- tasks "Forks" and "Chairs" activated here
    declare
        Dinner  : array (Num) of Philosopher ;
    begin-- All " Dinner (1..Num)" tasks activated here
        Dinner (1) .Birth ("Hegel",   2.0* years);
        Dinner (2) .Birth ("Kant",    1.0 * years);
        Dinner (3) .Birth ("Plato",   3.0 * years);
        Dinner (4) .Birth ("Pascal", 1.5 * years);
        Dinner (5) .Birth ("Marx",    2.5 * years);
    end;   -- Block waits for all "Dinner (1..Num)" tasks to terminate
end Dining Philosophers ; - subprogram waits for "Forks" and "Chairs" tasks to terminate
```

Figure 2. Ada task declarations example

Declared or *static tasks* depend on the Master who created them. Allocated or *dynamic tasks* depend on the Master containing the corresponding access type. Control does not leave a Master until all its depending tasks have terminated. In general, a Master terminates if it is completed and it has no dependent tasks or all its dependent tasks have terminated. The semantics of completions are somewhat more complicated. A Master is complete if any of the following holds: it has reached the last executable statement of its body, it has raised an unhandled exception, it has finished executing an exception handler, or it has executed a RETURN statement (for subprogram Masters), a RETURN, EXIT or GOTO statement (for block Masters), or a TERMINATE statement (for task Masters). These notions are illustrated in Figures 2 and 3.

Tasks interaction

Tasks interact with one another via a non-symmetric mechanism whereby tasks agree to ‘meet at a given place’, specified by the called task, and controlled by a self-enforcing delay protocol of tasks waiting on other tasks. That is, each task agrees to enter a ‘busy waiting’ loop if it needs to wait and it decides by itself when to leave this loop. Inter-task synchronization and communication occur at explicitly specified *rendezvous* points when a connection is made between two tasks, at which time information may be exchanged as specified by the called task interface. The asymmetry in the rendezvous has been noted by several authors [12,13] and it refers to

```
separate (Dining_Philosophers )
task body Forks is . . .
  State : array (Num) of Status := (others => Free);
  Left   : constant array (Num) of Num :- (5, 1, 2, 3, 4);
begin)
  loop  select
       accept Put_Down (F : Fork) do State (F):=  Free; State (Left(F)) := Free; end ;
       or when State (1) = Free and State (Left(1) ) = Free
       => accept Pick_Up (1) do State (1) :- In_Use; State (Left(1) ) := In_Use; end ;
       or when State (2) = Free and State (Left(2)) = Free      ...
       or  terminate; -- makes task complete when selected
       end  select ;
  end loop;
end Forks ;

separate (Dining_Philosophers )
task body Chairs is . . .
begin
  loop  select
       when Next_Chair <= Num' LAST  => accept Give (Me: out aChair ) do . . .
       or when Next_Chair >= Num'FIRST '> accept Here_is (My : aChair) do . . .
       or  terminate; -- makes task complete when selected
       end  select;
  end  loop;
end Chairs ;

separate (Dining_Philosophers )
task body Philosopher is
  type Stages is (Unborn, Hungry, Eating, thinking, Starved, Dead);
  DoB        : TIME; -- date of birth
  DoD        : DURATION :- 0.0; -- date of death
  Status     : Stages := Unborn; . . .
begin
  PUT_LINE ("A philosopher was conceived at " & INTEGER'IMAGE (INTEGER (SECONDS (CLOCK)) 1);
  accept Birth (Me  : Name; Life_Time : DURATION := 0.0) do . . . end Birth;
  Chairs.Give (Me => My_Chair);
  PUT_LINE (My_Name & " got chair # " & INTEGER'IMAGE (My_Chair) ) ; . . .
  loop Age := CLOCK - DoB;
       case Status is
       when Unborn     => PUT LINE ("**ERROR**") ;  raise TASKING_ERROR;
       when Thinking => PUT_LINE (My_Name & " Thinking");  Status := Hungry;
                 delay DURATION (DoD / 100 ); -- better use random num. generator
       when Hungry     => PUT_LINE (My_Name & " Hungry") ;
            select
                 Forks.Pick_Up (My_Chair);  Status := Eating;
            or delay DURATION (DoD) ;       Status := Starved;
            and  select;
       when Eating     => PUT_LINE (My_Name &  " Eating"); delay DURATION (DoD / 80);
                 Forks.Put_Down (My_Chair);  Status := Thinking;
       when Dead => PUT_LINE (My_Name & "Dead");
                 Forks.Put_Down (My_Chair); Chairs. Here_is (My_Chair);
                 exit; -- exits the loop !
       when Starved => PUT_LINE (My_Name &  " Starved to death!"); Status := Dead;
       end case;
       if Age >= DoD
       then Status := Dead; PUT_LINE (My_Name &  " Died of natural Causes "); end if;
  end loop;
  PUT_LINE (My_Name & "  buried ") ; -- task completed here !
end Philosopher ;
```

Figure 3. Ada task body example

the fact that only the caller knows the identity of the rendezvous partner. The callee must accept unidentified callers based upon an established protocol, and selective acceptance of a particular task is thus prevented. The semantics of the various tasking statements are discussed below; the *Ada Language Reference Manual (LRM)* or any of several textbooks [14,15] may be consulted for a more complete discussion.

Ada supports an explicit task communication mechanism in the form of an essentially procedural interface between exactly two tasks at a time. A task can call entries of another task. Upon accepting such calls a connection is established between the two tasks. A *task entry* defines a communication path and flow of data between the task defining it and any other calling task (see Figure 4). An entry may be called from the 'main program' since it is considered a subprogram called from some environment task. Entry calls are queued FIFO for each corresponding entry declaration. Hardware interrupts are also treated as entry calls by associating with it an *address representation clause;* such entries can still be called directly by other tasks! Like subprograms, entries can be overloaded; furthermore, most of the rules applicable for procedure declarations and procedure calls also apply to entry declarations and entry calls.

An entry's meaning is established by one or more accept statements inside the body of the called task. A special 'critical section' of the accept code defines a sequence of actions to be executed in mutual exclusion during a rendezvous. Control may only leave the rendezvous by reaching its end, executing a return statement, or raising an exception. Accepting calls may be fully synchronous or asynchronous, i.e.

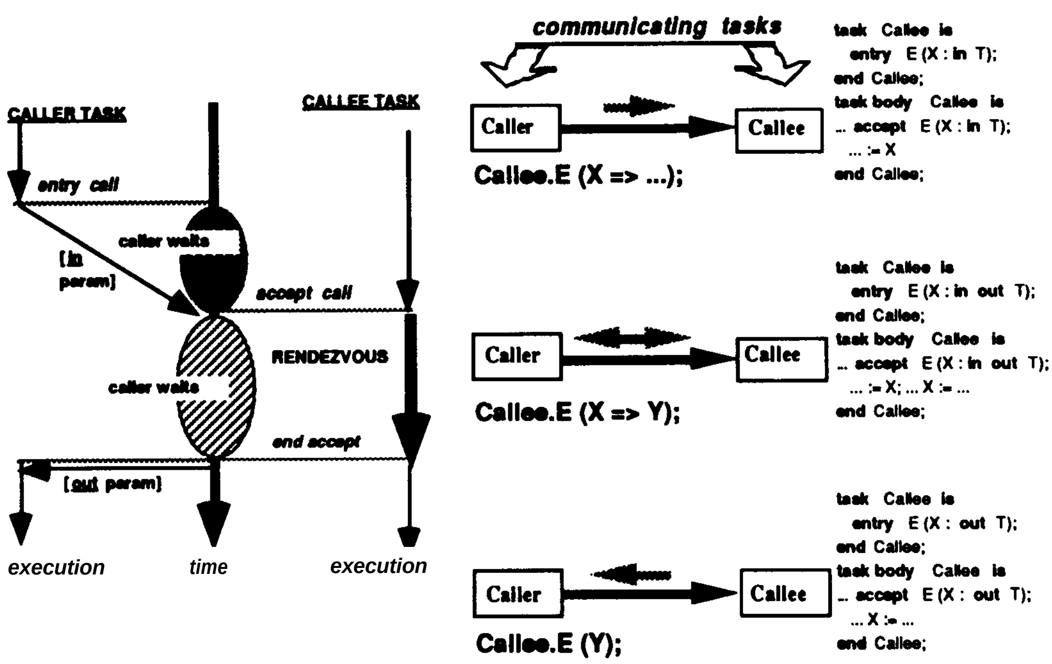

Figure 4. Entries and simple rendezvous

deterministic or non-deterministic. Alternative accepts within a select statement are non-deterministic.

The Ada rendezvous provides for many variations. Let us consider first the case in which a calling task executes an entry call and the callee task executes a corresponding accept. This *simple rendezvous* is illustrated in Figure 4. If the entry call precedes in time the accept statement, the 'caller' is queued indefinitely until the accept is made. If the accept statement precedes the entry call, the 'callee' is indefinitely blocked until an entry call is made. At the time an entry call is accepted, any parameters of mode 'in' or 'in out' are read-in and the called task executes the rendezvous code while the calling task waits for its conclusion. At the conclusion of the rendezvous, any updated parameters of mode 'out' or 'in out', are passed back to the calling task which is then removed from the entry queue, and its execution permitted to proceed concurrently again.

Other variations of the simple rendezvous are possible both on the side of the caller and on the side of the callee. On the one hand, the calling task can make conditional and timed calls. The *conditional entry call*, specified by an else alternative in a select statement, allows the code associated with this alternative to be selected if the call is not accepted 'immediately', i.e. a check is made to see if a corresponding accept has been executed, in which case the rendezvous takes place, and if not the call is cancelled. A *timed entry call*, specified by a delay alternative in a select statement, works in a similar fashion; the associated code is selected if the call is not accepted within at least the specified time period, the rendezvous takes place otherwise.

On the other hand, the accepting task can have various forms of a *selective wait*. In general, a selective wait means that a task may accept an entry call from a number of possible entries and optionally under certain circumstances specified as boolean expressions. These expressions, known as *guards* and associated with accepts statements, when evaluating to TRUE or when are not present, define an open accept alternative. When there are multiple open alternatives, one will be selected if the corresponding rendezvous can be immediately executed (the LRM does not specify which open alternative is selected). A *delay alternative* functions much as described above for the timed entry call. If an entry call has been made on the corresponding entry, the rendezvous is executed immediately; otherwise, the accepting task is blocked for a specified time period awaiting an entry call. If at the expiration of the delay time an entry call has not been made, the task proceeds without further delay and the select statement is completed. An else *alternative* behaves similarly to the conditional entry call above. A select statement may also contain a *terminate* alternative, which when selected will render the task complete.

As we can see, the semantics of the select statement is complex. For convenience, a summary of the select statement is listed in Table I.

KERNEL DESCRIPTION

The model we used to develop our kernel is based on a distributed kernel.[16] Some modifications were made to this message passing-based model and only those features required for a uniprocessor environment were included. The original model shows that nine exported kernel services are sufficient to handle the Ada task dynamics and inter-task communication. In addition, a procedure, StartTasks, was added in

Table I. Select statement semantics summary

1. There must be at least one accept alternative
2. At most one of the following may also appear
   (a) a terminate alternative, or
   (b) an else part, or
   (c) one or more delay statements.
3. A select alternative is open if either:
   (a) it does not have a guard (see below), or
   (b) it has a guard and the guard evaluates to true.
4. Before alternatives are considered, the following are evaluated in order:
   (a) guards (in an unspecified order), and
   (b) delay expression associated with open alternatives.
5. Open accept alternatives are considered first:
   (a) if rendezvous is possible, then one of the corresponding accept alternatives is selected arbitrarily (including more than one alternative for the same entry).
   (b) if rendezvous is not possible, and no else part does exist, the task is suspended until a call is made to an open accept alternative.
6. Open delay alternatives are considered next:
   if several open delay alternatives exist, the one with the smallest duration is selected, and if more than one exists with the same duration, one is chosen arbitrarily. (Negative delay durations are treated as zero).

The following additional rules further specify the semantics for the else part and the selection of terminate alternatives.

7. Selective wait with an else part:
   (a) else alternative selected if and only if no accept statement can be selected 'immediately'.
   (b) else alternative selected if no open alternatives exist. In fact it is an error if there are no open alternatives and no else; PROGRAM_ERROR is raised.
8. An open terminate alternative is selected if and only if
   (a) the task's Master is complete, and
   (b) any sibling (other tasks with same Master) is either terminated, or potentially terminated (i.e. waiting to terminate).

order to initially place the tasks into the ready queue. The small number of procedures is a result of combining all entry call and accept variants into only two 'control' units. The simple, conditional, and timed entry calls are combined into a single pruned entry call statement. The statement is implemented through a single procedure, EntryCall. Similarly, a pruned selective wait construct is used to represent the variations on the accept statement. Because of the greater complexity of the accept statement, two kernel procedures, AcceptBegin and AcceptEnd are required to implement the pruned selective wait. Additional kernel procedures are used to elaborate, activate, terminate, abort, delay, and transmit information on the task hierarchical structure. Task states and state transitions are shown in Figure 5; reference to this diagram and a study of the example and test programs presented later should clarify uncertainties in how the kernel is accessed and used.

Tasks are created in accordance with Ada semantics and task names assigned in the order of creation. The semantics of task elaboration and activation, defined earlier, are specified in the Ada LRM Section 9.3. Recall that the activation of a declared task starts after elaboration of the declarative part of the enclosing unit; formal elaboration will include a call to ARTK-M2 procedure ElaborateTask. This will be followed by a call to ActivateTask and, to indicate a child/parent relationship, by a call to ChildTask.

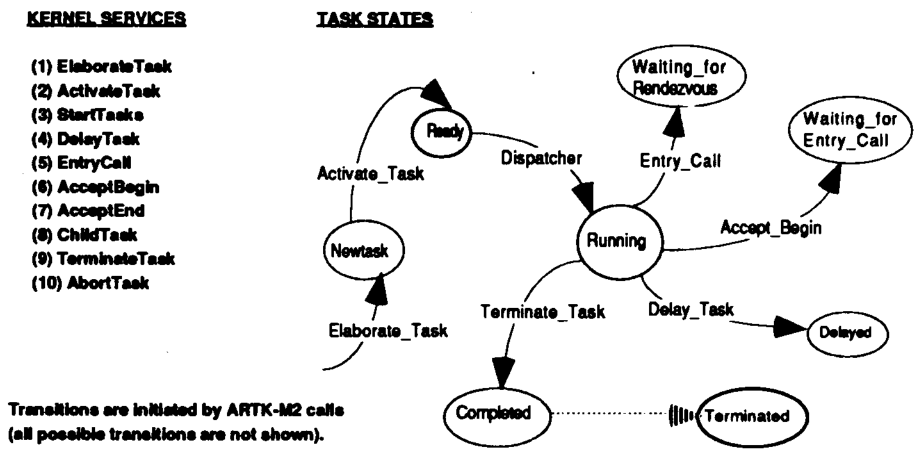

Figure 5. Task state transitions and kernel services

Tasking semantics and implementation details

The general implementation approach to the ARTK-M2 is as follows: each task is implemented in one-to-one correspondence with a Modula-2 coroutine. A virtual processor for each task is achieved by interleaved execution using the Modula-2 TRANSFER procedure. A non-pre-emptive scheduling algorithm is implemented with a context switch initiated each time ARTK-M2 services are requested. An alternative pre-emptive scheduling algorithm is implemented with a context switch initiated by time-slicing as well as by ARTK-M2 service requests. Access to ARTK-M2 services is exclusive y through the set of procedures listed above.

The ARTK-M2 has total responsibility for task generation, termination, scheduling, and inter-task communication and synchronization. Task termination and task abort are two issues of particular concern. It is not clear that all eventualities have been considered in the Ada LRM. The ARTK-M2 implements a termination algorithm based upon the LRM specifications, presented and discussed in a previous report; [17] this report also provides additional details and interface specification for each exported ARTK-M2 monitor procedure.

Data types and structures

The principal data structures used in the ARTK-M2 are four as follows (details found in Reference 17): (1) the task structure (TCB Table), (2) the ready queue, (3) the delay queue and (4) the busy queue. A task table, indexed by task id and implemented as an array of pointers to a record type defining a task control block (TCB), provides all information required for task management. The storage of entry parameters in the TCB is after the XINU operating system. [18] The ready queue is a simple priority queue list using an ARRAY [1.. Maximum Priority] of a FIFO queue type. The array index serves as the task priority and all tasks of equal priority are in the same FIFO queue. Although a more thorough discussion can be found in our earlier report, [17] it is noted here that implementation of the abstract type Queue proved to be crucial to the overall efficiency of the ARTK-M2. The baseline approach uses dynamically allocated nodes in a linked list structure. Subsequent testing showed this to be a particularly poor selection.

Task entry queues are also conventional FIFO queues using simple FIFO scheduling of the same type as the ready queue, since the Ada LRM specifies that priorities not be recognized in rendezvous scheduling. Simple delays are implemented using an ARRAY[1.. MaxTasks] of Time. The task name serves as the array index; the array entry is the task release time (release time = current time + delaytime.) The array is searched at every context switch and all tasks whose release time is less than or equal to the current time are inserted in the corresponding ready queue. The former is consistent with the LRM which requires delays to be 'at least' the duration specified. Linear search is used since the number of delayed tasks will probably be small. In addition to timing through tick counting, using the timer interrupt trapped and used to update the ARTK-M2 time, a variation was developed using calls to the system clock. This technique, while inherently more efficient, can be highly inaccurate since the ARTK-M2 time is updated only upon entry to the ARTK-M2 dispatcher.

WRITING ADA-LIKE TASKS IN MODULA-2

In what follows we discuss the mapping of Ada tasking programs into their Modula-2/ARTK-M2 counterparts. This process is illustrated by taking a concurrent Ada program and converting it into a Modula-2 equivalent program which makes calls to ARTK-M2 routines. Figure 6 shows the basic scenario for the translation process. Note that the translation may not be straightforward because all features of sequential Ada are not available in Modula-2. Each task, including the 'main program' is implemented as a Modula-2 coroutine. The implementation of the main Ada subprogram and environment task are reversed into a 'main' task (Task0) spawned from an

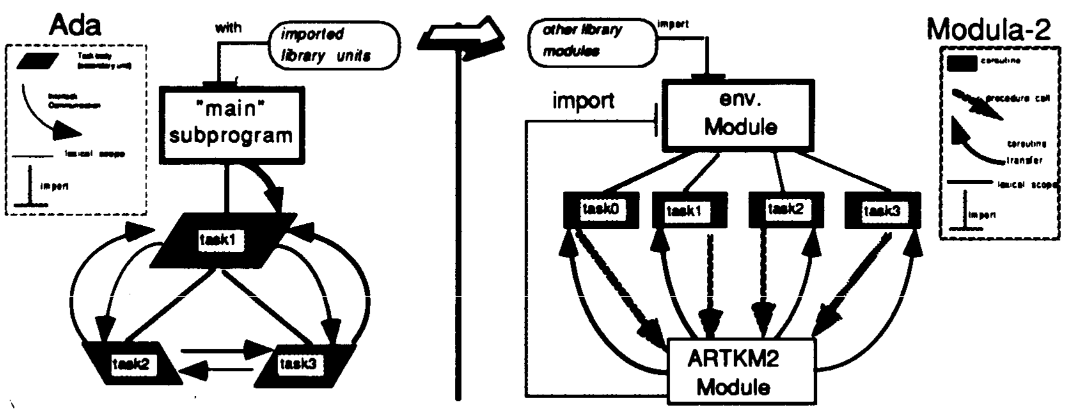

Figure 6. Mapping concurrent Ada into ARTK-M2 Modula-2 equivalents

environment ‘program’ (module EnvMod), respectively. Figure 7 shows details of a template module that can be used as a starting point.

A subset of the Ada Compiler Validation Capability (ACVC) test suite for tasks was processed in order to test the ARTK-M2 implementation. The ACVC tests are grouped according to the Ada LRM chapters, and into categories according to whether the test is testing compile-time, run-time or link-time features. The set of tests corresponding to tasking are found in chapter 9 of the LRM, and only those tests which are executable were picked; there were 200 plus tests, with an average length of 200 lines of Ada source code. An automatic translator generated the corresponding Modula-2/ARTK-M2 programs. This translation process is discussed next.

Automatic generation

The translation process was based on grammars augmented with procedural abstractions, attribute grammars, and fed to a parser generator tool (see Figure 8). To this end, a subset of the Ada syntax is augmented with translation routines representing semantic actions executed during the parsing process. These routines actually implement the transformation into Modula-2. The parser generator used was the Mystro Parser Generator (PARGEN) system. [19] It requires the semantics associated with the input grammar to be written in Pascal. The development of the translator proceeded as follows: since PARGEN tries to generate an LALR(l) parser, it was essential to maintain partial translations in order to collect inherited attributes and keep to a one pass translation of the Ada source. A good way to do this was the creation of intermediate data structures, called pseudo-files or psfiles for short, operated by a set of corresponding I/O routines.

Each record on the semantic stack in the parser contains a pointer to a user-defined structure, one part of which is a psfile. A frame on the stack is associated with each symbol in the RHS of a grammar rule. Therefore partial translations of previously scanned code can be combined with the current code and passed along. Three instances of psfiles were used. One such structure is used to keep parts of the source code currently being processed, but which have not been reduced as yet. Another psfile structure is used to contain Modula-2 declarations that are generated during the processing of Ada executable statements. Since these declarations are generated after the corresponding Ada declarative part has already been processed, they are saved in order to be emitted later when the declarative and executable parts are combined. For example, Modula-2 does not allow functions to return composite types, such types must be returned as pointers whose type declaration is placed on this psfile. The other psfile structure works in a similar way, but it contains executable code that must precede to the current statement being parsed.

A similar structure was defined for the symbol table. It contains information gleaned when the symbol is scanned. Consider for example a task entry-call, which syntactically looks identical to a procedure call and thus there is one grammar rule that serves to parse both statements; semantically, however, they are very different, with the task entry call requiring special handling to generate the appropriate calls to the ARTK-M2.

Translating the sequential Ada code was straightforward, given that the ACVC tests for tasking did not contain some of the more obscure sequential constructs,

```
MODULE EnvMod; (* This module is a template for converting Ada concurrent
 programs into Modula-2 programs with calls to ARTK-M2 service routines *)
IMPORT ARTKM 2 ; (*The following are available:(accessed with prefix ARTK.M2)
-- Constants
    InfiniteDelay, Infinite Time, Max Tasks, MaxEntryPerTask, TaskName,
    MaxChildren,  MaximumPriority, NumberOfEntries, PriorityNumber;
-- Types
    EntryName, Time, GuardArray, IndexArray, AcceptData, TaskAllocation, ,
    EntryData,  TaskError, AcceptRecord, TaskRecord, EntryRecord,
    ProcessPointer, SimulatedTime, CurrentTime;
-- Variables
    Elaborate Task, Activate Task, Delay Task, Start Tasks;
-- Procedures
    EntryCall, AcceptBegin, AcceptEnd, ChildTask, Terminate Task, AbortTask, Busy; *)
( *!!! Add other imported modules as required!!!* )

CONST TimeSlice = 5;        (*!!! Change as desired. !!!* )
( *!!! Add other constants as required. !!!* )

TYPE   Task0ParmsRec = record . . . end; ( *!!! Add for each task entry parameter. !!!* )
       Task0Entry = (Task0Entry1, . . .);      ( *!!! Add for each task entry. !!!* )
( *!!! Add other types as required. !!!* )

VAR    Task0Name  : ARTKM2 . TaskName;    ( *!!! Add for each task. !!!*)
       Task0Parms : Task0ParmsRec ;     ( *!!! Add for each task. !!!*)
( *!!! Add other variables as required. !!!*)
PROCEDURE  Task0; (* This procedure represents the Ada "main" program. *)
             (* this is a Modula-2 coroutine, and thus must not return * )
BEGIN ( * Main program logic here.  Permissible kernel calls are:
             ARTKM2.EntryCall/.DelayTask/.Busy/.TerminateTask/.AbortTask*)
END Task0;
PROCEDURE  Task 1 ; (* This procedure represents a typical Ada task. *)
      (* this is a Modula-2 coroutine, and thus must not return *)
BEGIN (* Main program logic here.  Possible kernel calls are:
             ARTKM2.EntryCall/.AcceptBegin/.AcceptEnd/.DelayTask/Busy
                      .Terminate  Task/.AbortTask;
       The following kernel cells CANNOT be made:
       ARTKM2.ElaborateTask/.ActivateTask/.ChildTask/.StartTasks
       (Ada requires parent task not be active until all of its children are
        active, thus these calls can only be made from the main module) *)
END Task1;
(*!!! Repeat above for the desired number of tasks. !!!*)
BEGIN (* Module EnvMod: represents the Ada environment task *)
   ARTKM2. ElaborateTask( Task0Name, Task0Name, Task0, 5000, 1, 0 );
   ARTKM2.ElaborateTask( Task1Name, Task0Name, Task1, 5000, 1, 0 );
       (* Repeat above for all tasks; use appropriate parameters *)
       (* Order is not important but Task O must be elaborated first. *)
   ARTKM2.ActivateTask( Task1Name );
       (* Activate all tasks that have no children with the above call. *)
       (* Then repeat for all tasks that have no grandchildren. Etc. *)
       (* This insures that no task starts running until all of its children
             are running.  Note that Task0 is not explicitly activated.
             This is done implicitly by ARTKM2. *)
   ARTKM2.ChildTask( Task0Name, Task1Name );
       (* Repeat above call as necessary *)
   ARTKM2.StartTasks
END EnvMod.
```

Figure 7. Template Modula-2 concurrent program

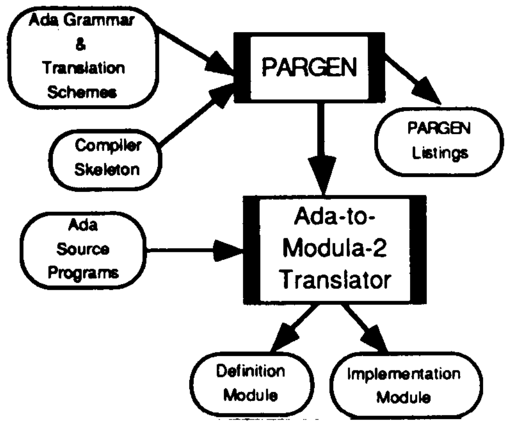

Figure 8. PARGEN elements

and only those constructs that were in the tests were addressed. Of the 200+ relevant tests, only about 80 tests were actually processed; most of the non-processed tests were testing tasking exceptions or using exceptions to implement the test objective; the current version of ARTK-M2 does not implement exceptions. The addition of exception handling facilities to Modula-2 programs has been discussed in the literature; exception handling will be incorporated in the kernel at a later stage. Although our goal was not to write a translator for Ada, we ended up processing many Ada constructs not directly related to tasking. One of the more interesting ones is the Ada block statement, which is translated into an anonymous procedure whose declaration is moved to the declarative section of the surrounding block and replaced by a call, as shown in Figure 9.

A task specification containing one or more entries, with or without parameters is translated as shown in Figure 10, with each entry becoming a record which contains

```
-- blocks:
begin  -- tasks "Forks" and "Chairs" ...
  declare
  Dinner: array (Num) of Philosopher;
begin  -- All "Dinner (1..Num)"  ...
  Dinner(1).Birth("Hegel",2.0*Years);
  Dinner(2).Birth("Kant", 1.0*Years);
  Dinner(3).Birth("Plato", 3.0*Years);
  Dinner(4).Birth("Pascal",1.5*Years);
  Dinner(5).Birth("Marx", 2.5*Years);
 end; -- Block waits ...
end Dining Philosophers;
```

=>

```
PROCEDURE  Main;
  VAR  ATHOME:  BOOLEAN;
  PROCEDURE  Anonymous1;
  BEGIN
     PhilosopherParms.Me   :=   "Hegel";
     Clear(PhilosopherParms.Life_Time);
     PhilosopherParms.
     Life_Time  [Minute]  :=  2.0;
     ARTKM2.EntryCall(Dinner[1],
       ORD(PhilosopherBirth)+1,ORD(1),
       ADR(PhilosopherParms),
       ARTKM2.INFINITEDELAY,   ATHOME);
     PhilosopherParms.Me   :=   "Kant";
     . . .
  END  Anonymous1;
BEGIN
  Anonymous1;
  ARTKM2.TerminateTask(  0  );
END  Main;
```

Figure 9.

a field for every parameter. For simplicity, even parameterless entries generate a record. All entries for a particular task are then combined into an enumeration type. Names for the various identifiers are constructed by catenating the task name and entry name with descriptive suffixes.

A task body with simple rendezvous generates the code in Figure 11.

A call to a task entry relies upon information previously saved when parsing the task specification, and further, it is necessary to check the symbol table to distinguish between a call to an entry and call to a procedure, since they can be syntactically identical. In the translation illustrated in Figures 12 and 13, the parameter in the call is an 'in' parameter, and so must be assigned to the proper field in the Parms record before the entry call. The first argument in the call to EntryCall is the task control block index. The second is the index of the entry, i.e. the first entry. The third argument is the address of the parameter record and the last specifies an infinite wait for an accept.

Elaboration, activation, and specification of parentchild relationships are the last things to happen before the tasks are explicitly started. See Figure 7, lines 40–50, for an illustration.

TEST RESULTS

Qualitative testing for implementation errors used sample test programs from the Ada Compiler Validation Capability. The C-tests for tasking consist of 206 separate tests, of these, less than half could be translated in a straight forward manner with little or no modification. A large portion of the tests (120) contain exception handlers which have no counter-part in Modula-2, nor are implemented in our kernel; fortunately most of these tests could be run after some modifications which had no significant impact on the quality of the test.

For quantitative testing we used a version of the kernel for which no time dependent activity is permitted. The principle quantitative testing sequence compared

```
-- task specifications:
task Forks is
  entry Pick_Up  (Fork); -- a family...
  entry Put_Down (F : Fork);
end Forks;
...
task type Philosopher is
  entry Birth (Me : Name;
         Life_Time : DURATION := 0.0);
end Philosopher ;
```

=>

```
TYPE  ForksParmsRec  =
        RECORD    F:  Fork; END;
TYPE  ForksParmsRecPtr  =
        POINTER  TO  ForksParmsRec;
VAR  ForksParms:  ForksParmsRec;
TYPE  ForksEntry=
        (ForksPick_Up1,ForksPick_Up2,
        ForksPick_Up3,ForksPick_Up4,
        ForksPick_Up5,ForksPut_Down);
VAR  TASKForks:  ARTKM2.TaskName;
TYPE  PhilosopherParmsRec  =
   RECORD   Me:  Name;
              Life_Time:  Duration;  END;
TYPE  PhilosopherParmsRecPtr  =
        POINTER  TO  PhilosopherParmsRec;
VAR
   PhilosopherParms:PhilosopherParmsRec;
TYPE
   PhilosopherEntry=(PhilosopherBirth);
TYPE  TASKPhilosopher=ARTKM2.TaskName;
VAR Dinner:  ARRAY  Num  OF
              TASKPhilosopher;
```

Figure 10.

-- simple rendezvous:

```
task body Philosopher is
...
begin
  PUT_LINE (...
  accept Birth (Me : Name;
     Life_Time : DURATION := 0.0) do
     My_Name (1..Me'LENGTH) := me;
     Status := Hungry;
     DoD := Life_Time;
     PUT_LINE (My_Name & "was born");
  end Birth;
...
```

=>

```
PROCEDURE  Philosopher;...
BEGIN...
  ARTKM2.AcceptBegin(Guards,
  ARTKM2.INFINITEDELAY,
    PhilosopherParmsPtr,   SELECTOR);
    Me  := PhilosopherParmsPtr^.Me;
    Life_Time  :=
       PhilosopherParmsPtr^.Life_Time;
    My_Name:=Me;    Status:=Hungry;
    DoD:=Life_Time;
    Concat(My_Name,"  was  born",
       STRING3);
    WriteString(STRING3);  WriteLn;
  ARTKM2.AcceptEnd(SELECTOR,
       PhilosopherParmsPtr);...
END  Philosopher;
```

Figure 11.

--Ada entry call

```
task body Philosopher is
...
begin
  ...
     Chairs.Give (Me => My_Chair);
```

=>

```
ARTKM2.EntryCall(TASKChairs,
       ORD(ChairsGive)+1,   ORD(1),
       ADR(ChairsParms),
       ARTKM2.INFINITEDELAY,   ATHOME);
My_Chair  :=  ChairsParms.Me;
```

Figure 12.

```
separate(Dining_Philosophers)
task body Forks is
...
begin
   loop  select
   accept Put_Down (F : Fork) do
      Forks_State (F) := Free;
      Forks_State (Left(F)) := Free;
      end ;
   or when Forks_State (1) = Free and
      Forks_State (Left(1)) = Free
   => accept Pick_Up (1) do
      Forks_State (1) := In_Use;
      Forks_State (Left(1)) := In_Use;
      end ;
   ...
   or  terminate;
   end  select;
   end  loop;
end  Forks;
```

```
PROCEDURE  Forks;   ...
BEGIN
  Forks_State[1]:=Free;   Left[1]:=5;...
  LOOP
     WaitTime  :=  ARTKM2.INFINITEDELAY;
     FOR  iGuards  :=  1  TO  HIGH(Guards)
     DO  Guards[iGuards]  :=  0;  END;
     IF  (Forks_State[1]=Free)  AND
        (Forks_State[Left[1]]=Free)
     THEN  Guards  [ORD(ForksPick_Up1)+1]
           :=  -1;  END;  ...
     Guards[ORD(ForksPut_Down)+1]:=   -1;
     ARTKM2.AcceptBegin(Guards,
     WaitTime,  ForksParmsPtr,SELECTOR);
     CASE  SELECTOR  OF
       1:  Forks_State[1]:=In_Use;
       Forks_State[Left[1]]:=In_Use;
       ARTKM2.AcceptEnd(SELECTOR,
       ForksParmsPtr);
     |2:  ...
     END  (*  CASE  *);  END  (*  LOOP  *);
  ARTKM2.TerminateTask(  0  );
END  Forks;
```

Figure 13.

the rendezvous approach with that of monitors for mutual exclusion purposes. The profiler tool used intercepts the18·2 Hz timer interrupt and examine the processor registers and program counter. Comparison of these values with addresses obtained from the linker map were sufficient to infer the percentage of time spent on each module.

Examination of the results shows that nearly 79 per cent of the run time was spent in the STORAGE, QUEUE, and PQUEUE modules. The STORAGE module implements dynamic storage allocation; the only program elements making significant use of dynamic memory allocation are QUEUE and PQUEUE for the linked list implementation of task entry queues and ready queue. As remarked earlier, the dynamic queue implementation was a particularly poor choice. It is also noted that nearly 19 per cent of the run time was spent inside the ARTK-M2 and only 2 per cent in procedures representing the actual applications code.

Significance

The fractional time spent in productive work is not in itself a reliable indicator of inherent inefficiency. A more significant number is the ratio of the rendezvous and semaphore overheads. The theoretical lower bound for the rendezvous/semaphore overhead ratio is then 3:1, somewhat less than the 3·4:1 measured value. The value in excess of 3·0 represents the ARTK-M2 contribution to the total overhead. The overhead ratio should be task insensitive and only weakly dependent upon other factors such as machine architecture and compiler efficiency. It is interesting to note that even with a zero overhead ARTK-M2 and infinitely fast queueing operations, the Ada rendezvous has twice the overhead of a comparable semaphore implementation of these examples.

SUMMARY AND FUTURE RESEARCH

Ada syntax provides constructs for specifying actions to be performed by more than one task. These actions are executed in sequence, and several of these sequences may be in progress at the same time. The language treats synchronization and communication on an equal footing; the rendezvous mechanism is intended to support both. A run-time kernel is needed to implement overlapped or interleaved concurrency. A reasonably complete implementation of a run-time kernel supporting Ada tasking has been discussed. The semantic rules of Ada tasking allows for several different implementations of a supporting run-time kernel. Some maybe more suited to certain applications than others. A message-based model for a uniprocessor environment was implemented as a Modula-2 module. Individual Ada tasks are represented as coroutines; full support for task creation, synchronization, communication, scheduling, and termination is provided through 10 exported ARTK-M2 procedures. All options of the Ada rendezvous including conditional entry and accept calls, task priorities, multiple entries, parameter passing, and the delay and terminate alternatives of the select statement are provided. ARTK-M2 testing, though incomplete, included qualitative and quantitative test programs to reveal implementation errors and estimate rendezvous overhead costs. The latter used a simple problem to compare current semantics with Ada rendezvous behavior and proposed extensions.

The ARTK-M2 can be a useful tool in the study of real-time control systems, serve as a baseline for implementing alternatives seeking improved efficiency, and when properly extended, form an integral part of a distributed Ada run-time kernel in a multiprocessing environment. The efficiency issues discussed here and elsewhere are critical and alternate implementations must be explored. Recently proposed additional semantics to take care of priority scheduling problems [20] are being implemented in ARTK-M2. A restructuring of the kernel following the object-oriented philosophy is also being considered; this version of the kernel is planed to be written in Oberon. [21]

And finally, some known limitations and potential problem areas of the implementation include: (1) The handling of task exceptions arising during task activation (LRM 9.3) is not implemented. (2) As mentioned earlier, the problems of task termination (LRM 9.4) are significant and all eventualities may not be covered by the LRM. The task termination algorithm used in the ARTK-M2 should therefore be critically studied. (3) Task and Entry Attributes (LRM 9.9) have not been included but inclusion of these features should not be difficult. (4) The Task Abort Statement (LRM 9.10) has only been implemented up to Paragraph 5.

REFERENCES

1. *Reference Manual for the Ada Programming Language,* ANSI/Military Standard MIL-STD-1815A, US Department of Defense, January 1983.
2. N. Wirth, *Programming in Modula-2,* 2nd edn, Springer-Verlag, 1983.
3. P. N. Hilfinger, ‘Implementation strategies for Ada tasking idioms’, *Proceedings of the AdaTec Conference on Ada,* October 1982, p. 26.
4. A. N. P. Habermann and I. R. Nassi, ‘Efficient implementation of Ada tasks’, Technical Report *CMU-CS-80-103,* Carnegie-Mellon University, 1980.
5. W. Eventoff, D. Harvey and R. J. Price, ‘The rendezvous and monitor concepts: is there an efficiency difference?’, *Proceedings of ACM-SIGPLAN Symposium on the Ada Programming* Language, December 1980, p. 156.
6. W. Hoyer, ‘Intertask communication realized with an interrupt mechanism’, *‘Proceedings of the AdaTec Conference on Ada,* 1985.
7. A. Jones and A. Ardo, ‘Comparative efficiency of different implementations of the Ada rendezvous’, *Proceedings of the Ada-TEC Conference on Ada,* October 1982, p. 212.
8. T. P. Baker and G. A. Riccardi, ‘Ada tasking: from semantics to efficient implementation’, *IEEE* Software, **2,** (2), 34, (1985).
9. D. A. Fisher and R. M. Weatherly, ‘Issues in the design of a distributed operating system for Ada’, IEEE Computer, 19, (5), 38, (1986).
10. *Ada9X Project Report Mapping Document,* Draft, Office of the Under Secretary of Defense for Acquisition, Washington, D.C. 20301, U.S.A, February 1991.
11. J. L. Díaz-Herrera, ‘Hierarchical modular diagrams: an approach to describe the static software structure’ to appear in *IEEE Software.*
12. J. Welsh and A. Lister, ‘A comparative study of task communication in Ada’, Software—Practice & Experience, **11,** 257, (1981).
13. N. Francez and S. A. Yemini, ‘Symmetric intertask communication’, *ACM Transactions on Programming Languages and Systems,* **7,** (4), 622, (1985).
14. N. Cohen, *Ada as a Second Language,* McGraw-Hill, 1986.
15. J. G. P. Barnes, Programming In Ada, 3rd edn, Addison-Wesley, 1989.
16. R. M. Weatherly, ‘A message-based kernel to support Ada tasking’, *Proceedings of the Conference on Ada Applications and Environments,* IEEE Computer Society, October 1984, p. 136.
17. J. L. Díaz-Herrera and R. Graft, ‘Ada Multitasking for Modula-2’, George Mason University, Fairfax, Virginia, Department of Computer Science, TR, 3–90, 1990.
18. D. Comer, Operating System Design, *The XINU Approach,* Prentice-Hall, 1984, p. 95.
19. R. Conings, ‘Pargen: an LR parser generator’, College of William and Mary, 1985.
20. L. Sha and J. B. Goodenogh, ‘Real-time scheduling theory and Ada’, *IEEE Computer,* April 1990, pp. 53-62.
21. N. Wirth, ‘The programming language Oberon’, *Software—Practice and Experience,* **18,** 671–690 (1988).

APPENDIX: DINING PHILOSOPHERS EXAMPLE

--Dining Philosophers Ada Example

```
with CALENDAR, TEXT_IO;
use CALENDAR, TEXT_IO;
with Random;use Random;
procedure Dining_Philosophers is
   subtype Num is POSITIVE range 1..5;
   subtype Fork is Num;
   subtype Chair is Num;
   subtype Name is STRING;
   Years : constant := 60;

   task Forks is
      entry Pick_Up (Fork; ; --a family of entries
      entry Put_Down (F : Fork) ;
   end Forks;

   task Chairs is
      entry Give (Me_A: out Chair );
      entry Here_is (My : Chair);
   end Chairs ;

   task type Philosopher is
      entry Birth (Me : Name; Life_Time : DURATION := 0.0);
   end Philosopher ;

   task body Forks is separate;
   task body Chairs is separate;
   task body Philosopher is separate;

begin
   declare
      Dinner : array (Num) of Philosopher ;
   begin
      Dinner (1) .Birth ("Hegel", 2.0 l Years);
      Dinner (2) .Birth ("Kant", 1.0 * Years);
      Dinner (3) .Birth ("Plato", 3.0 “* Years);
      Dinner (4) .Birth ("Pascal", 1.5 * Years);
      Dinner (5) .Birth ("Marx", 2.5 * Years);
   end;
end Dining_Philosophers:

separate (Dining_Philosophers )’
task body Forks is
   type State is (In_Use, Free);
   Forks_State : array (Num) of State := (others => Free);
   Left  : constant array (Num) of Num := (5, 1, 2, 3, 4);
begin
   loop select
      accept Put_Down (F : Fork) do
         Forks_State (F) := Free;
         Forks_State (Left(F)) := Free;
      end Put_Down ;
      or when Forks_State (1) = Free and Forks_State (Left(1) ) = Free
      => accept Pick_Up (1) do
         Forks_State (1) := In_Use;
         Forks_State (Left(l)) := In_Use;
      end Pick_Up:
      or when Forks-State (2) = Free and Forks_State (Left(2)) = Free
      => accept Pick_Up (2) do
```

```
            Forks_State (2) := In_Use;
            Forks_State (Left(2)) := In_Use;
         end Pick_Up;
         or when Forks_State (3) = Free and Forks_State (Left(3)) = Free
         => accept Pick_Up (3) do
            Forks_State (3) := In_Use;
            Forks_State (Left(3)) := In_Use;
         end Pick_Up;
         or when Forks_State (4) = Free and Forks_State (Left(4)) = Free
         => accept Pick_Up (4) do
            Forks_State (4) := In_Use;
            Forks_State (Left(4)) := In_Use;
         end Pick_Up:
         or when Forks_State (5) = Free and Forks_State (Left(5)) = Free
         => accept Pick_Up (5) do
            Forks_State(5) := In_Use;
            Forks_State  Left (5)  := In_Use;
         end Pick_Up;
         or terminate;
         end select;
      end loop;
   end Forks ;

   separate (Dining_Philosophers)
   task body Chairs is
      Next_Chair : POSITIVE := 1;
   begin
      loop select
         when Next_Chair <= NUM'LAST
         => accept Give (Me_A: out Chair ) do
               Me_A := Next_Chair ;
               Next_Chair := Next_Chair + 1;
             end Give :
         or when Next_Chair >= NuM'FIRST
         => accept Here is (My : Chair) do
               Next_Chair := Next_Chair - 1:
            end Here_is ;
         or terminate;
         end select;
      end loop:
   end Chairs ;

   separate (Dining_Philosophers)
   task body Philosopher is
      type Stages is (Unborn, Hungry, Eating, thinking, Starved, Dead);
      DOB            : TIME; -- date of birth
      DoD            : DURATION := 0.0; -- date of death
      Age            : DURATION := 0.0;
      Status         : stages := Unborn;
      My_Name          STRING (1..10) := (1..10 => ' ');
      My_Chair       : Chair;
   begin
      PUT_LINE ("A philosopher was conceived at " &
         INTEGER'IMAGE (INTEGER(SECONDS (CLOCK) )));
      accept Birth (Me : Name; Life_Time : DURATION := 0.0) do
         My_Name (1. .Me'LENGTH) := me;
         Status := Hungry;
         DoD := Life_Time;
```

-- Dining Philosophers Ada Example

```
         PUT_LINE (My_Name & " was born") ;
      end Birth;
      Chairs .Give (Me_A => My_Chair);
      PUT_LINE (My_Name & " got chair # " & INTEGER'IMAGE (My_Chair));
      DoB := CLOCK;
      loop
         Age := CLOCK - DoB;
         case Status is
         when UnBorn       => PUT LINE ("**ERROR**") ;
                              raise TASKING_ERROR;
         when Thinking     => PUT_LINE (My_Name & " Thinking") ;
                              delay DURATION ( RandomReal*20.0 );
                              Status := Hungry;
         when Hungry       => PUT_LINE (My_Name & " Hungry") ;
                              select
                                 Forks.Pick_Up (My_Chair);
                                 Status := Eating;
                              or delay DURATION ( RandomReal*10.0 );
                                 Status := Starved;
                              end select;
         when Eating       => PUT_LINE (My_Name & " Eating"*) ;
                              delay DURATION ( RandomReal*5.0 );
                              Forks.Put_Down (My_Chair); Status := Thinking;
         when Dead         => PUT_LINE (My_Name & " Dead") ;
                              Forks.Put_Down (My_Chair);
                              Chairs.Here_is (My_Chair);
                              exit:
         when Starved      => PUT LINE (My_Name & " Starved to death! ") ;
                              Status := Dead;
         end case;

         if Age >= DoD then
            Status := Dead:
            PUT_LINE (My_Name & " Died of natural causes");
         end if;
      end loop;
      PUT LINE (My_Name & " Buried");
   end Philosopher ;
```

-- dining Philosophers Ada Example: results

```
---------------------------------------------------------------
A philosopher was conceived at  36373
A philosopher was conceived at  36373
A philosopher was conceived at  36373
A philosopher was conceived at  36373
A philosopher was conceived at  36373
Hegel      was born
Kant       was born
Plato      was born
Pascal     was born
Marx       was born
Hegel      got chair # 1
Hegel      Hungry
Hegel      Eating
Kant       got chair # 2
Kant       Hungry
Plato      got chair # 3
Plato      Hungry
Plato      Eating
Pascal     got chair # 4
```

-- Dining Philosophers Ada Example: results

```
21   Pascal    Hungry
22   Marx      got chair # 5
23   Marx      Hungry
24   Hege 1    Thinking
25   Plato     Thinking
26   Marx      Eating
27   Kant      Eating
28   Harx      Thinking
29   Pascal    Eating
30   Kant      Thinking
31   Pascal    Thinking
32   Pascal    Hungry
33   Pascal    Eating
34   Kant      Hungry
35   Kant      Eating
36   Pascal    Thinking
37   Marx      Hungry
38   Marx      Eating
39   Hegel     Hungry
40   Kant      Thinking
41   Marx      Thinking
42   Hegel     Eating
43   Plato     Hungry
44   Plato     Eating
45   Marx      Hungry
46   Hege 1    Thinking
47   Marx      Eating
48   Plato     Thinking
49   Pascal    Hungry
50   Hegel     Hungry
51   Marx      Thinking
52   Plato     Hungry
53   Hegel     Eating
54   Pascal    Eating
55   Pascal    Thinking
56   Plato     Eating
57   Hegel     Thinking
58   Plato     Thinking
59   Kant      Hungry
60   Kant      Eating
61   Plato     Hungry
62   Plato     Starved to death!
63   Plato     Dead
64   Plato     Buried
65   Pascal    Hungry
66   Pascal    Eating
67   Kant      Thinking
68   Pascal    Thinking
69   Kant      Hungry
70   Kant      Eating
71   Kant      Thinking
72   Hegel     Hungry
73   Hegel     Eating
74   Marx      Hungry
75   Hegel     Thinking
76   Pascal    Hungry
77   Marx      Eating
78   Marx      Thinking
79   Pascal    Eating
80   Pascal    Thinking
```

--Dining Philosophers Ada Example: results

```
 81   Hegel     Hungry
 82   Hegel     Eating
 83   Hegel     Thinking
 84   Kant      Hungry
 85   Kant      Eating
 86   Kant      Thinking
 87   Marx      Hungry
 88   Marx      Eating
 89   Pascal    Hungry
 90   Marx      Thinking
 91   Pascal    Eating
 92   Pascal    Thinking
 93   Pascal    Hungry
 94   Pascal    Eating
 95   Pascal    Thinking
 96   Hegel     Hungry
 97   Hegel     Eating
 98   Pascal    Hungry
 99   Pascal    Eating
100   Hegel     Thinking
101   Kant      Died of natural causes
102   Kant      Dead
103   Kant      Buried
104   Pascal    Thinking
105   Marx      Hungry
106   Marx      Eating
107   Marx      Thinking
108   Hegel     Hungry
109   Hegel     Eating
110   Hegel     Thinking
111   Marx      Hungry
112   Marx      Eating
113   Pascal    Hungry
114   Marx      Thinking
115   Pascal    Died of natural causes
116   Pascal    Dead
117   Pascal    Buried
118   Marx      Hungry
119   Marx      Eating
120   Marx      Thinking
121   Hegel     Hungry
122   Hegel     Eating
123   Hegel     Thinking
124   Marx      Hungry
125   Marx      Eating
126   Marx      Thinking
127   Hegel     Hungry
128   Hegel     Eating
129   Hegel     Thinking
130   Marx      Hungry
131   Marx      Eating
132   Marx      Thinking
133   Hegel     Hungry
134   Hegel     Died of natural causes
135   Hegel     Dead
136   Hegel     Buried
137   Marx      Hungry
138   Marx      Eating
139   Marx      Thinking
140   Marx      Hungry
141   Marx      Died of natural causes
142   Marx      Dead
143   Marx      Buried
```

--Dining Philosophers Modula-2 translation

```
------------ ------- ------ ------- ------- ------------- ------------------- .
------------ ------- ------------------------------- ------------------- .

MODULE Dining_Philosophers;
FROM Calendar IMPORT Date, GetMachineDate, DeltaDate;
FROM DurationOps IMPORT Duration, UnitSet, GreaterOrEqual, Unit, Clear;
FROM Terminal IMPORT WriteString, WriteLn;
FROM AdaAttributes IMPORT INTEGER_IMAGE:
FROM Strings IMPORT ConCat;
FROM SYSTEM IMPORT ADR;
FROM Break IMPORT EnableBreak;
IMPORT ARTKM2;
FROM AdaTypes IMPORT NATURAL, POSITIVE, STRING;
FROM TimeDate IMPORT Time, GetTime, TimeToString;
FROM Random IMPORT RandomReal;

TYPE Num= POSITIVE[1. .5];
TYPE Fork= Num;
TYPE Chair= Num;
TYPE Name= STRING;
CONST Years=60.0;
CONST TicksPerSecond-20.0;
VAR I:POSITIVE;

TYPE ForksParmsRec =
RECORD
   F: Fork;
END ;
TYPE ForksParmsRecPtr = POINTER TO ForksParmsRec;
VAR ForksParms: ForksParmsRec;
TYPE ForksEntry= (ForksPick_Up1, ForksPick_Up2, ForksPick_Up3,
                  ForksPick_Up4, ForksPick_Up5, ForksPut_Down) ;
VAR TASKMain: ARTKM2.TaskName;
VAR TASKForks: ARTKM2.TaskName;

TYPE ChairsParmsRec =
RECORD
      Me_A: Chair;
      My: Chair;
END ;
TYPE ChairsParmsRecPtr = POINTER TO ChairsParmsRec;
VAR ChairsParms: ChairsParmsRec;
TYPE ChairsEntry= (ChairsGive, ChairsHere_is);
VAR TASKChairs: ARTKM2.TaskName;

TYPE PhilosopherParmsRec =
RECORD
      Me: Name;
      Life_Time: Duration;
END ;
TYPE PhilosopherParmsRecPtr = POINTER TO PhilosopherParmsRec;
VAR PhilosopherParms: PhilosopherParmsRec;
TYPE PhilosopherEntry= (PhilosopherBirth) ;
TYPE TASKPhilosopher=ARTKM2 .TaskName;
VAR Dinner: ARRAY Num OF TASKPhilosopher;

PROCEDURE Forks;
   VAR ForksParmsPtr: ForksParmsRecPtr;
```

-- Dining Philosophers Modula-2 translation

```
      VAR F: Fork;
      VAR Guards: ARTKM2. GuardArray;
      VAR iGuards: CARDINAL;
      VAR WaitTime: ARTKM2. Time;
      VAR SELECTOR: CARDINAL;
      TYPE State= (In_Use,Free);
      VAR Forks_State: ARRAY Num OF State;
      VAR Left: ARRAY Num OF Num;
   BEGIN
      Forks_State[1] :=Free;
      Forks_State[2] :=Free;
      Forks_State[3] :=Free;
      Forks_State[4] :=Free;
      Forks_State[5] :=Free;
      Left[1] :=5;
      Left[2] :=l;
      Left[3] :=2;
      Left[4] :=3;
      Left[S] :-4;
      LOOP
            WaitTime := ARTKM2. INFINITEDELAY;
            FOR iGuards := 1 TO HIGH(Guards) DO Guards[iGuards] := 0; END;
            IF (Forks_State[1] =Free) AND (Forks_State[Left [1]]=Free) THEN
               Guards[ORD(ForksPick_Up1) +1] := -1;
            END ;
            IF (Forks_State[2] -Free) AND (Forks_State[Left [2]]=Free) THEN
               Guards [ORD(ForksPick_Up2) +1] := -1;
            END ;
            IF (Forks_State[3]=Free) AND (Forks_State[Left [3]]=Free) THEN
               Guards [ORD(ForksPick_Up3) +1] := -1;
            END ;
            IF (Forks_State[4]=Free) AND (Forks_State [Left[4]]=Free) THEN
               Guards [ORD(ForksPick_Up4) +1] := -1;
            END ;
            IF (Forks_State[5]=Free) AND (Forks_State[Left [5]]=Free) THEN
               Guards[ORD(ForksPick_Up5) +1] := -1;
            END ;
            Guards [ORD(ForksPut_Down) +1] := -1;
            ARTKM2.AcceptBegin (Guards, WaitTime, ForksParmsPtr, SELECTOR);
            CASE SELECTOR OF
            1:
               Forks_State[1] :=In_Use;
               Forks_State [Left[1]] :=In_Use;
               ARTKM2.AcceptEnd(SELECTOR, ForksParmsPtr) :
            |
            2:
               Forks_State[2] :=In_Use;
               Forks_State [Left[2]] :=In_Use;
               ARTKM2.AcceptEnd(SELECTOR, ForksParmsPtr) ;
            |
            3:
               Forks_State[3] :=In_Use;
               Forks_State[Left [3]] :=In_Use;
               ARTKM2.AcceptEnd(SELECTOR, ForksParmsPtr);
            |
            4:
               Forks_State[4] :=In_Use;
               Forks_State[Left [4]] :=In_Use;
```

```
               ARTKM2.AcceptEnd(SELECTOR, ForksParmaPtr);
            |
            5:
               Forks_State[5] :=In_Use;
               Forks_State[Left [5] ] :=In_Use;
               ARTKM2.AcceptEnd(SELECTOR, ForksParmsPtr);
            |
            6:
               F:=ForksParme.Ptr^ .F;
               Forks_State[F] :=Free;
               Forks_State[Left [F]] :=Free;
               ARTKM2.AcceptEnd(SELECTOR, ForksParmsPtr);
            END (* CASE *);
      END (* LOOP *);
      ARTKM2.TerminateTask( 0 );
   END Forks;

   PROCEDURE Chairs;
      VAR ChairsParmsPtr: ChairsParmsRecPtr;
      VAR Me_A; Chair;
      VAR My: Chair;
      VAR Guards: ARTKM2.GuardArray;
      VAR iGuards: CARDINAL;
      VAR WaitTime: ARTKM2.Tirne;
      VAR SELECTOR: CARDINAL;
      VAR Next_Chair:POSITIVE;
   BEGIN
      Next_Chair :=1;
      LOOP
         WaitTime := ARTKM2. INFINITEDELAY;
         FOR iGuards := 1 TO HIGH(Guards) DO Guards[iGuards] := 0; END;
         IF Next_Chair<=MAX(Num) THEN
            Guards [ORD(ChairsGive) +1] := -1;
         END ;
         IF Next_Chair>=MIN(Num) THEN
            Guards [ORD(ChairsHere_is) +1] := -1;
         END ;
         ARTKM2.AcceptBegin (Guards, WaitTime, ChairsParmsPtr, SELECTOR);
         CASE SELECTOR OF
         1:
            Me_A:=Next_Chair;
            Next_Chair :=Next_Chair+ 1;
            ChairsParmsPtr^.Me A := Me_A;
            ARTKM2.AcceptEnd (SELECTOR,ChairsParmsPtr) ;
         |
         2:
            My :=    ChairsParmsPtr^ .My;
            Next_Chair:=Next_Chair- 1;
            ARTKM2.AcceptEnd(SELECTOR, ChairsParmsPtr);
         |
         3:
            ARTKM2.TerminateTask( 0 );
         END (* CASE *);
      END (* LOOP *);
      ARTKM2.TerminateTask( 0 );
   END Chairs;

   PROCEDURE Philosopher;
```

-- Dining Philosophers Modula-2 translation

```
    VAR ATHOME : BOOLEAN ;
    VAR PhilosopherParmsPtr: PhilosopherParmsRecPtr;
    VAR Me: Name;
    VAR Life_Time: Duration:
    VAR Guards: ARTKM2.GuardArray;
    VAR iGuards: CARDINAL;
    VAR WaitTime: ARTKM2.Time;
    VAR SELECTOR: CARDINAL;
    TYPE Stages= (Unborn,Hungry,Eating, Thinking, Starved,Dead) ;
    VAR DoB:Date;
    VAR DoD:Duration;
    VAR Age:Duration;
    VAR Status:Stages;
    VAR My_Name:STRING;
    VAR My_Chair:Chair;
    STRING1 :ARRAY[0. .16] OF CHAR;
    STRING2 :ARRAY[0. .48] OF CHAR;
    STRING3 :ARRAY[0..14] OF CHAR;
    STRING4 :ARRAY[0. .18] OF CHAR;
    STRING5 :ARRAY[0. .4] OF CHAR;
    STRING6 :ARRAY[0. .23] OF CHAR;
    STRING7 :ARRAY[0. .14] OF CHAR;
    STRING8 :ARRAY[0. .14] OF CHAR;
    STRING9 :ARRAY[0. .14] OF CHAR;
    STRING10 :ARRAY[0. .14] OF CHAR;
    STRING1l :ARRAY[0. .23] OF CHAR;
    STRING12 :ARRAY[0. .28] OF CHAR;
    STRING13 :ARRAY[0. .12] OF CHAR;
    CLOCK:Date;
 BEGIN
    GetMachineDate (CLOCK);
    INTEGER_IMAGE (CLOCK.second, STRING1);
    Concat("A philosopher was conceived at ",STRING1,STRING2);
    WriteString (STRING2) ;WriteLn:

    FOR iGuards := 1 TO HIGH(Guards) DO Guards[iGuards] := O; END;
    Guards[ORD(PhilosopherBirth) +1] := -1;
    ARTKM2.AcceptBegin (Guards, ARTKM2. INFINITEDELAY,
       PhilosopherParmsPtr, SELECTOR);
       He := PhilosopherParmsPtr^ .Me;
       Life_Time := PhilosopherParmsPtr^ .Life_Time;
       My_Name:=Me:
       Status:=Hungry;
       DoD:=Life_Time:
       Concat(My_Name ," was born",STRING3) ;
       WriteString(STRING3) ;WriteLn;
    ARTKM2.AcceptEnd(SELECTOR, PhilosopherParmsPtr) ;

    ARTKM2.EntryCall (TASKChairs, ORD(ChairsGive)+1, ORD(1),
       ADR(ChairsParms), ARTKM2. INFINITEDELAY, ATHOME);
    My_Chair := ChairsParms.Me_A;
    Concat(My_Name," got chair # ",STRING4);
    INTEGER_IMAGE ( My_Chair, STRING5);
    Concat(STRING4, STRINFG5,STRING6) ;
    WriteString (STRING6);WriteLn;
    GetMachineDate (DoB);
    LOOP
       GetMachineDate (CLOCK);
```

```
        DeltaDate (DOB, CLOCK, Unit Set(Second), Age) ;

        CASE Status OF
        Unborn:
           WriteString ("**ERROR**"); WriteLn;
        |
        Thinking:
           Concat(My_Name," Thinking", STRING7) ;
           WriteString (STRING7);WriteLn;
           ARTKM2.DelayTask(O, TRUNC(RandomReal ()* TickspersSecond*20.0));
           Status:=Hungry:
        |
        Hungry:
           Concat(My_Name," Hungry",STRING8 );
           WriteString (STRING8);WriteLn;
           CASE My_Chair OF
           1:
              ARTKM2.EntryCall (TASKForks, ORD(ForksPick_Up1) +1, ORD(1),
                 ADR(ForksParms), TRUNC(RandomReal ()* TicksPerSecond*10.0),
                 ATHOME ) ;
              IF ATHOME THEN
                 Status:=Eating;
              ELSE
                 Status:=Starved;
              END (* IF *);
           |
           2:
              ARTKM2.EntryCall (TASKForks, ORD(ForksPick_Up2) +1, ORD(1),
                 ADR(ForksParms), TRUNC(RandomReal ()* TicksPerSecond*10.0),
                 ATHOME) ;
              IF ATHOME THEN
                 Status:=Eating;
              ELSE
                 Status:=Starved;
              END (* IF *);
           |
           3:
              ARTKM2.EntryCall (TASKForks, ORD(ForksPick_Up3) +1, ORD(l),
                 ADR(ForksParms), TRUNC(RandomReal ()* TicksPerSecond*10.0),
                 ATHOME) ;
              IF ATHOME THEN
                 Status:=Eating;
              ELSE
                 Status:=Starved;
              END (* IF *);
           |
           4:
              ARTKM2.EntryCall (TASKForks, ORD(ForksPick_Up4) +1, ORD(1),
                 ADR(ForksParms) , TRUNC(RandomReal () *TicksPerSecond*10 .0),
                 ATHOME) ;
              IF ATHOME THEN
                 Status:=Eating;
              ELSE
                 Status:=Starved;
              END (* IF *);
           |
           5:
              ARTKM2.EntryCall (TASKForks, ORD(ForksPick_Up5) +1, ORD(1),
```

```
                  ADR(ForksParms) , TRUNC (RandomReal ( ) *TicksPerSecond* 10. O) ,
                  ATHOME ) ;
               IF ATHOME THEN
                  Status :=Eating;
               ELSE
                  Status:=Starved;
               END (* IF *);
            END (* CASE *);
         |
         Eating:
            Concat(My_Name, " Eating",STRING9) ;
            WriteString (STRING9);WriteLn;
            ARTXt42.DelayTask(O, TRUNC(RandomReal ()* TicksPerSecond*5 .0));
            ForksParms.F := My_Chair;
            ARTKM2.EntryCall (TASKForks, ORD(ForksPut_Down) +1, ORD(1),
               ADR(ForksParms), ARTKM2. INFINITEDELAY, ATHOME);
            Status:=Thinking;
         |
         Dead:
            Concat(My_Name," Dead",STRING10);
            WriteString (STRINGI10 ;WriteLn;
            ForksParms.F := My_Chair:
            ARTKM2.EntryCall (TASKForks, ORD(ForksPut_Down) +, ORD(1),
               ADR(ForksParms), ARTKM2. INFINITEDELAY, ATHOME);
            ChairsParms.My := My_Chair;
            ARTKM2.EntryCall (TASKChairs, ORD(ChairsHere_is) +1, ORD(1),
               ADR(ChairsParms), ARTKM2. INFINITEDELAY, ATHOME);
            EXIT;
         |
         Starved:
            Concat(My_Name, "Starved to death!",STRING11);
            WriteString (STRING11) ;WriteLn;
            Status:=Dead;
         END (* CASE *);

         IF GreaterOrEqual (Age,DoD, Second) THEN
            Status:=Dead;
            Concat(My_Name, "Died of natural causes", STRINGl2);
            WriteString (STRING12) ;WriteLn;
         END ;
      END (* LOOP *);
      Concat(My_Name," Buried", STRING13) ;
      WriteString (STRING13);WriteLn;
      ARTKM2.TerminateTask( 0 );
   END Philosopher;

   PROCEDURE Main;
      VAR ATHOME: BOOLEAN;
      PROCEDURE Anonymous:
      BEGIN
         PhilosopherParms .Me := "Hegel";
         Clear (PhilosopherParms.Life_Time);
         PhilosopherParms .Life_Time [Minute]:= 2.0;
         ARTKM2.EntryCall (Dinner[1],
            ORD (PhilosopherBirth) +1,
            ORD(1),
            ADR(PhilosopherParms),
            ARTKM2.INFINITEDELAY,
```

```
            ATHOME ) ;
         PhilosopherParms.Me := "Kant";
         Clear(PhilosopherParms  .Life_Time);
         PhilosopherParma .Life_Time [Minute] := 1.0;
         ARTKM2.EntryCall  (Dinner[2],
            ORD(PhilosopherBirth) +1,
            ORD(1),
            ADR(PhilosopherParms),
            ARTKM2.INFINITEDELAY,
            ATHOME ) ;
         PhilosopherParms.Me := "Plato";
         Clear(PhilosopherParms .Life  Time);
         PhilosopherParms.Life_Time  [Minute]  := 3.0;
         ARTKM2.EntryCall  (Dinner[3],
            ORD(PhilosopherBirth) +1,
            ORD(1),
            ADR(PhilosopherParms) ,
            ARTKM2. INFINITEDELAY,
            ATHOME ) ;
         PhilosopherParms.Me := "Pascal";
         Clear (PhilosopherParms .Life Time);
         PhilosopherParms .Life_Time [Minute] := 1.5;
         ARTKM2.EntryCall  (Dinner[4],
            ORD (PhilosopherBirth) +1,
            ORD(1),
            ADR(PhilosopherParms),
            ARTKM2.INFINITEDELAY,
            ATHOME) ;
         PhilosopherParms.Me := "Marx";
         Clear(PhilosopherParms  .Life_Time);
         PhilosopherParms .Life_Time [Minute] := 2.5;
         ARTKM2.EntryCall  (Dinner[5],
            OFfD (PhilosopherBirth) +1,
            ORD(1),
            ADR(PhilosopherParms) ,
            ARTKM2. INFINITEDELAY,
            ATHOME) ;
      END  Anonymous1;
   BEGIN
      Anonymous;
      ARTKN2.TerminateTask( 0 );
   END Main;

   BEGIN
    (* Task Initialization *)
   EnableBreak;
   ARTKM2.ElaborateTask (TASKMain, TASKMain, Main, 5000, 1, 0);
   FOR 1:=1 TO 5 DO
      ARTKM2.ElaborateTask (Dinner[I], TASKMain, Philosopher, 5000, 1, 1);
   END (* FOR *);
   FOR 1:=1 TO 5 DO
      ARTKM2.ActivateTask (Dinner[I] ) ;
   END (* FOR *);
   FOR 1:=1 TO 5 DO
      ARTKM2.ChildTask (TASKMain, Dinner[I]) ;
   END (* FOR *);
   ARTKM2.ElaborateTask (TASKChairs, TASKMain, Chairs, 5000, 1, 2) ;
   ARTKJ42.ActivateTask (TASKChairs) ;
```

-- Dining Philosophers Modula-2 translation

```
ARTKM2.ChildTask (TASKMain, TASKChairs) ;
ARTKM2.ElaborateTask (TASKForks, TASKMain, Forks, 5000, 1, 6);
ARTKM2.ActivateTask(TASKForks) ;
ARTKM2.ChildTask(TASKMain, TASKForks);
ARTKM2.ChildTask(TASKMain, TASKMain);
ARTKM2.StartTasks();
END Dining_Philosophers.
```

-- Dining Philosophers Modula-2 translation: results

```
-----------------------------------------------------------------------

A philosopher was conceived at     51
A philosopher was conceived at     51
A philosopher was conceived at     51
A philosopher was conceived at     51
A philosopher was conceived at     51
Hegel was born
Kant was born
Hegel got chair #      1
Hegel Hungry
Plato was born
Kant got chair #      2
Kant Hungry
Hegel Eating
Pascal was born
Plato got chair #      3
Plato Hungry
Marx was born
Pascal got chair #      4
Pascal Hungry
Plato Eating
Marx got chair #      5
Marx Hungry
Plato Thinking
Pascal Eating
Pascal Thinking
Hegel Thinking
Marx Eating
Kant Eating
Pascal Hungry
Kant Thinking
Marx Thinking
Pascal Eating
Marx Hungry
Pascal Thinking
Marx Eating
Marx Thinking
Kant Hungry
Kant Eating
Plato Hungry
Kant Thinking
Plato Eating
Kant Hungry
Marx Hungry
Marx Eating
```

-- Dining Philosophers Modula-2 translation: results

```
47   Kant Starved to death!
48   Kant Dead
49   Kant Buried
50   Plato Thinking
51   Hegel Hungry
52   Marx Thinking
53   Hegel Eating
54   Hegel Thinking
55   Pascal Hungry
56   Pascal Eating
57   Plato Hungry
58   Pascal Thinking
117  Marx Eating
118  Plato Hungry
119  Plato Eating
120  Plato Thinking
121  Marx Thinking
122  Marx Hungry
123  Marx Eating
124  Plato Hungry
125  Plato Eating
126  Marx Thinking
127  Marx Hungry
128  Marx Eating
129  Plato Thinking
130  Marx Thinking
131  Hegel Hungry
132  Hegel Died of natural causes
133  Hegel Dead
134  Hegel Buried
135  Marx Hungry
136  Marx Eating
137  Plato Hungry
138  Plato Eating
139  Marx Thinking
140  Plato Thinking
141  Marx Hungry
142  Marx Died of natural causes
143  Marx Dead
144  Marx Buried
145  Plato Hungry
146  Plato Eating
147  Plato Thinking
148  Plato Hungry
149  Plato Died of natural causes
150  Plato Dead
151  Plato Buried
152   Ready Queue is empty. . .
153   Deadlock has occurred. .
154
```

-- Dining Philosophers Modula-2 translation: results

```
 59  Plato Eating
 60  Marx Hungry
 61  Marx Eating
 62  Plato Thinking
 63  Hegel Hungry
 64  Marx Thinking
 65  Hegel Eating
 66  Plato Hungry
 67  Plato Eating
 68  Hegel Thinking
 69  Marx Hungry
 70  Marx Eating
 71  Plato Thinking
 72  Marx Thinking
 73  Plato Hungry
 74  Plato Eating
 75  Pascal Hungry
 76  Pascal Starved to death!
 77  Pascal Dead
 78  Pascal Busied
 79  Plato Thinking
 80  Hegel Hungry
 81  Hegel Eating
 82  Plato Hungry
 83  Plato Eating
 84  Plato Thinking
 85  Marx Hungry
 86  Hegel Thinking
 87  Marx Eating
 88  Plato Hungry
 89  Plato Eating
 90  Marx Thinking
 91  Plato Thinking
 92  Plato Hungry
 93  Plato Eating
 94  Plato Thinking
 95  Hegel Hungry
 96  Hegel Eating
 97  Hegel Thinking
 98  Marx Hungry
 99  Marx Eating
100  Marx Thinking
101  Marx Hungry
102  Marx Eating
103  Marx Thinking
104  Plato Hungry
105  Plato Eating
106  Plato Thinking
107  Hegel Hungry
108  Hegel Eating
109  Marx Hungry
110  Hegel Thinking
111  Marx Eating
112  Marx Thinking
113  Hegel Hungry
114  Hegel Eating
115  Marx Hungry
116  Hegel Thinking
```