Incorporating Cyclic Task Behaviour into Ada Tasks

Patrick Bernardi
Research School of Engineering
The Australian National University
patrick.bernardi@anu.edu.au

Abstract

While Ada incorporates comprehensive support for real-time systems, it is missing a common programming abstraction used in the design of these systems: cyclic tasks. Without the abstraction, users resort to solutions that introduce unnecessary complexity into their programs — making it harder to read and write what should be a simple primitive. The omission of a cyclic task abstraction appears due to an aversion to implementing a too higher-level abstraction within Ada, and the perceived complexity and limitations brought to the language as a result. This paper will demonstrate the aversion is unwarranted and present an extension to Ada's existing task type to provide support for a cyclic task abstraction. The extension will show an emphasis on clarity, structure and implementability, while minimising the impact on the existing language.

1 Introduction

A gap exists between the design and specification of tasks in a real-time system and their implementation in Ada. These tasks are inherently cyclic in nature, yet there is no simple way to express these tasks — referred to as cyclic tasks in this paper — and their properties in Ada.

Through the system design processes, each cyclic task is specified with a set of timing, release and constraint properties. To implement them, Ada provides a rich set of real-time facilities that allows users to build sophisticated cyclic tasks to these specifications. Ada can build cyclic tasks with different release mechanisms; build tasks that detect and respond to missed deadlines and over-running their execution bounds; and safely mix aperiodic tasks with periodic and sporadic tasks through the sharing of CPU budgets. The problem Ada has in realising these cyclic tasks lies with the word *build*: the real-time facilities provided by Ada — while powerful — are low-level system primitives that form the building blocks the user uses to assemble the task themselves — a processes repeated for each new cyclic task type. This approach is not desired as it is error prone, not a good use of time, and makes the system software harder to read and maintain. In essence, the current Ada approach to cyclic tasks is akin to the C approach to a simple, entry-less protected object: C does not provide direct support; instead the user has to roll their own solution using the locking primitives provided by an external library.

Andy Wellings and Alan Burns in [1] presented a solution to the building problem by way of a library of real-time utilities. These reusable utilities are defined in a standards-like manner, alleviating the work required to build a cyclic task. While a considerable improvement on building cyclic tasks by hand, the real-time utilities library approach does not close the gap between the specification of a real-time task and its implementation: a cyclic task is still built from a collection of separate objects with most of its properties hidden inside its body. In addition, the library makes use of Ada features that are not supported on systems that use the Ravenscar Profile or that prohibit the use of dynamic dispatching — restrictions commonly used by real-time systems.

Table 1. An example cyclic task specification.

Priority	15
Cycle behaviour	Periodic
Period	1 s
Phase	200 ms
Relative deadline	500 ms
Miss deadline handling	Call procedure
Cycle execution budget	100 ms
Execution budget overrun handling	Call procedure

This paper presents a new approach for implementing cyclic tasks in Ada. Instead of building a library providing cyclic task components, this paper extends the syntax and semantics of Ada tasks to include support for cyclic tasks, providing a high-level cyclic task abstraction usable on high-integrity real-time systems — removing the gap between the design of a task and its implementation in Ada.

2 Real-Time Utilities Library

Before presenting the new Ada cyclic task behaviour, first a look at the real-time utilities library approach to implementing cyclic tasks — and how it does not fully meet the needs of real-time system users. As a brief introduction, the real-time utilities library breaks a cyclic task into three components:

- A task state object that contains the state of the task and the user code that operates on the state.
- A release mechanism used to release each task cycle. This component also detects missed deadlines and execution overruns.
- A real-time task, which implements the cyclic behaviour.

An additional fourth component provides support for aperiodic tasks sharing common CPU budget.

Each component in the library provides a selection of types that offer differing behaviour that — once assembled — provides the user a cyclic task with the desired properties. Underlying this library of real-time components is tagged records and type interfaces, used to minimise the number of types each component provides. For the task state object, the library provides three abstract tagged records to support the three types of cyclic tasks: periodic, sporadic and aperiodic. Users then create their own task state type by extending one of these tagged records to include their own state and actions; all without needing to touch the state associated with the cyclic operations of the task. Release mechanisms types provide different cycle release behaviour and levels of support for missed deadlines and execution time overruns and are provided in the form of protected objects. Finally, the cyclic behaviour of the task and the task itself is provided through one of several real-time tasks, which differ in their termination semantics: these tasks bring together the task, its state and release mechanism.

Illustrating the use of the real-time utilities library, here is a cyclic task with the specification in Table 1:

```
type My_Cyclic_Task_State is new Periodic_Task_State with record
      ... task state ...
end record;

procedure Initialise (S : in out My_Cyclic_Task_State);
procedure Code (S : in out My_Cyclic_Task_State);
procedure Deadline_Miss (S : in out My_Cyclic_Task_State);
procedure Overrun (S : in out My_Cyclic_Task_State);
```

```
My_Task_State   : aliased My_Cyclic_Task_State_Type;
My_Task_Release : aliased Periodic_Release (S => My_Task_State'Access);
My_Task         : Periodic_Release_With_Deadline_Miss_And_Overrun
                    (S         => My_Task_State'Access,
                     R         => My_Task_Release'Access,
                     Init_Prio => Default_Priority);

procedure Initialise (S : in out My_Cyclic_Task_State) is
begin
   S.Pri               := 15;
   S.Period            := Seconds (1);
   S.Phase             := Milliseconds (200);
   S.Relative_Deadline := Milliseconds (500);
   S.Execution_Time    := Milliseconds (100);
   ... initialise task code ...
end Initialise;

procedure Code (S : in out My_Cyclic_Task_State) is
begin
   ... cyclic code ...
end Code;

procedure Missed_Deadline (S : in out My_Cyclic_Task_State) is
begin
   ... missed deadline handler ...
end Missed_Deadline;

procedure Overrun (S : in out My_Cyclic_Task_State) is
begin
   ... cycle execution overrun handler ...
end Overrun;
```

This example highlights the structure the real-time utilities library uses to implement cyclic tasks and how the three components provided by the library link together. Together with abstracting out the cyclic code, the real-time utilities library provides a clarity and structure that does not exist when implementing the cyclic task using the low-level tasking primitives provided by Ada directly.

Despite this, the real-time utilities library approach falls short of meeting the needs of users:

1. Loss of tasking abstraction

Ada provides a unique tasking abstraction, encapsulating the state of the task and the code that operates on the state within a single programming unit. A public specification and a private body provide this abstraction using the same approach as used by packages and protected objects. A program benefits from this abstraction as it provides a new scope from which declarations relevant only to the task can be made, resulting in clearer and easier to maintain programs. Additionally, the scope ensures a task cannot access another task's internal state.

The real-time utilities library loses this tasking abstraction since the task type is only used to implement the cyclic behaviour. Declaration of a task's state now occurs outside the task body — allowing another task within the same scope read and modify the former task's state. Non-state task-specific declarations under the library have to be defined by repeating the same declarations in both the initialisation and cyclic procedures, or by exposing these declarations in a scope outside the task — neither options aiding the development of maintainable software.

2. Decomposing the cyclic task abstraction

The real-time utilities library approach requires the user to think of cyclic tasks as three separate objects, even though conceptually the cyclic task is one object. Thus, the library requires the user to use a lower programming abstraction than what they have designed around. An implication of this is a simple periodic task with no deadline or execution time monitoring is more complex to write using the real-time utilities library.

3. A cyclic task's specification is not clear

Recreating the specifications of the periodic task in Table 1 from the above implementation is not straightforward. Discovery of cyclic behaviour of the task requires the reader to examine what type the state was derived from, or alternatively examine the release mechanism used. The other properties of the task require the reader to delve into the body of the initialisation procedure to discover their values. This is problematic as it mixes the specification of the task with the code initialising the task's user state — harming readability — and requires the task's implementation just to divine the timing properties of the task.

Having to define the timing properties of a task as assignment statements in the initialisation procedure also mean these properties cannot be constant. This opens the door for potential bugs on hard real-time systems — where the timing properties of tasks are not meant to change by design as it complicates the system's schedulability analysis — if the user accidentally modifies these properties in the cyclic task code.

4. Unusable on some systems

While simplifying the real-time utilities library, the extensive use of Ada's tasking and object-oriented features prevents the library from being used on high-integrity systems or real-time systems that make use of the Ravenscar Profile. Specifically, the prohibition of dynamic dispatching on many high integrity systems precludes the library's use in these systems, while the implementation of the release mechanisms relies on protected objects with multiple entries, ability to requeue entry calls and presence of execution time timers — all parts of the Ada tasking run-time that are forbidden under the Ravenscar Profile.

3 Incorporating Cyclic Tasks into Ada

Preserving Ada's existing tasking abstraction and providing support for a cyclic task abstraction that considers a cyclic task as a single object requires syntactical support from Ada. To minimise the impact on the language, this paper presents support for cyclic tasks in Ada through a new optional cyclic section inserted into the body of an Ada task. Semantics of an Ada task only change when this cyclic section is present, the behaviour of which is tailored through a set of new task properties. Interaction of Ada tasks — with or without the cyclic section — with the rest of the language remains unchanged.

3.1 Task Body

To provide the cyclic section of the task body, the formal syntax of the task body becomes:

```
task_body ::=
   task body defining_identifier
        [aspect_specification] is
     declarative_part
   begin
     sequence_of_statements
   [cycles
     sequence_of_statements]
   [exception
     exception_handler
    {exception_handler}]
   end [task_identifier];
```

The new keyword **cycles**[1] denotes the cyclic section of the task. Without this section the semantics of a task remains unchanged. With the statements, the semantics of the task becomes:

1. The first *sequence_of_statements* forms the sequential or initialisation section of the task — allowing the user to setup the task before the cyclic execution begins. Execution of this section occurs after the activation of the task.
2. The *sequence_of_statements* after the **cycles** keyword forms the cyclic section of the task, with its statements executed cyclically — an iteration of the statements is known as a task cycle.
3. A cycle release point exists at the start of the cyclic section. A new task cycle will commence when the task is released by the run-time. The task's *Cyclic_Behavior* property determines the release of a task cycle (see Section 3.3).
4. Once a task completes a task cycle, it shall block from executing until the release of the next task cycle.
5. If a new task cycle can commence while an existing task cycle has not completed, the new task cycle will commence at the completion of the current task cycle. There can only be one such pending task cycle (for example: if a currently executing aperiodic task is a task cycle and multiple release events occur, only one task cycle will be executed in response to those multiple release events).
6. Absolute deadline for a task cycle is the release time of the task cycle plus the task's relative deadline.
7. At the cycle release point, a task's execution budget replenishes to the value of the task's execution budget property.
8. The earliest the first task cycle can commence is the sum of: the system defined global start time, the partition's *System_Start_Offset* property and the *Cyclic_Phase* property of the task.
9. Under normal operation, the task will not terminate.
10. Exception handlers will handle any exception raised from either *sequence_of_statements*.

3.2 System Start Time

Regardless of the cyclic behaviour of a task, the release of the first task cycle will not occur before a run-time defined start time. This is to provide a common start time for all tasks and allow tasks to complete their initialisation before the system commences its cyclic operations. A configuration pragma *System_Start_Offset* is provided to delay the run-time defined start time if more time is need to initialise the system:

```
pragma System_Start_Offset (Ada.Real_Time.Time_Span)
```

The default value of *System_Start_Offset* is `Ada.Real_Time.Time_Span_Zero`.

3.3 Task Properties

Ada defines one task property in the core language and a further four in the Real-Time Annex:

- *Storage_Size*
- *Priority*
- *CPU*
- *Dispatching_Domain*
- *Absolute_Deadline*

---

1. Keyword **cycles** was chosen in part due to:
   - *cycle:* already used by the Ada Reference Manual as a parameter to a number of Numerics subprograms
   - *loop:* accidentally omitting or adding the **end loop** label to a task body would change its semantics.

Support for the semantics of the cyclic task section requires a further ten new task properties. A new package `Ada.Cyclic_Tasks` provides the types used by these properties and the functions to read them. Representation aspects of the same name may be specified on the task type (including the anonymous type of a single task declaration) to set these properties. Properties — excluding *Cycle_Behavior* — can also be set dynamically at run-time using the procedures defined in the `Ada.Cyclic_Tasks.Dynamics` package. Placing these dynamic procedures in a separate package allows a user to enforce static cyclic task properties by using:

```
pragma Restrictions (No_Dependence => Ada.Cyclic_Tasks.Dynamics)
```

Applying this restriction has the advantage of forcing the timing specification of the task to be specified on the declaration of the task and not inside its body — making the timing specification clear in the user's code. Appendix A provides the specification of each package.

Cycle_Behavior

Type: `Ada.Cyclic_Tasks.Behavior` | *Default:* `Yield`

Defines how the run-time releases a task so it can commence a new task cycle. There are four different release behaviours:

Yield – A yielding cyclic task has to yield the processor before a new task cycle can commence (i.e. the release point acts as a task dispatching point).

Periodic – New task cycles may commence at fixed periodic intervals, determined by the *Cycle_Period* of the task.

Aperiodic – A new task cycle may commence after a call to `Ada.Cyclic_Tasks.Release_Task`.

Sporadic – A new task cycle may commence after a call to `Ada.Cyclic_Tasks.Release_Task`, but no earlier than the last cycle release time plus the *Cycle_Period* of the task.

Unlike the remaining properties, the *Cycle_Behavior* property is not modifiable at run-time since there is little justification for a task to change its release behaviour. Consequently, the compiler or run-time may use different data-structures to implement the different cycle behaviours. An attribute `T'Cyclic_Behavior` exists to return the cyclic behaviour for the prefix task.

Cycle_Period

Type: `Ada.Real_Time.Time_Span` | *Default:* `Ada.Real_Time.Time_Span_Last`

For periodic tasks, it is the time between task cycle release points. For sporadic tasks, the cycle period is the earliest time from the last task cycle release point that the task may be released. It has no effect for yielding or aperiodic tasks.

Cycle_Phase

Type: `Ada.Real_Time.Time_Span` | *Default:* `Ada.Real_Time.Time_Span_Zero`

Applies to all cyclic tasks and defines the offset from the system start time from which the first task cycle of a task may commence.

Relative_Deadline

Type: `Ada.Real_Time.Time_Span` | *Default:* `Ada.Real_Time.Time_Span_Last`

The real time a task cycle has to complete from its release point.

Execution_Budget

Type: `Ada.Real_Time.Time_Span` | *Default:* `Ada.Real_Time.Time_Span_Last`

The amount of execution time a task cycle can consume.

Deadline_Response, Budget_Response

Type: `Ada.Cyclic_Tasks.Event_Response` | *Default:* `No_Response`

Defines how the run-time reacts to a task cycle missing its deadline or exceeding its execution budget:

No_Response – The run-time does not react.

Handler – The run-time calls a protected procedure handler.

Abort_Cycle – The run-time aborts the current task cycle and the task waits for the next cycle release point.

Raise_Exception – The run-time raises `Deadline_Missed_Error` or `Execution_Budget_Exhausted_Error` in the task that caused the exception to be raised.

Deadline_Handler, Budget_Handler

Type: `Ada.Cyclic_Task.Response_Handler`

The protected procedure called by the run-time if a task cycle misses its deadline or exceeds its execution budget respectively. The handler is passed the `Task_Id` of the task that caused the handler to be called, allowing a handler to be shared amongst a set of tasks.

Execution_Server

Type: `Ada.Execution_Servers.Execution_Server`

The execution server property specifies the execution server object that will provide the shared CPU budget for the task to use.

3.4 Execution Servers

Execution servers allow a set of tasks to share a single execution budget and enable the scheduling of aperiodic tasks amongst periodic and sporadic tasks. The `Ada.Execution_Servers` and `Ada.Execution_Servers.Dynamics` packages provide the library support for execution servers.

Due to the variety of execution server implementations (see [2]), the `Ada.Execution_Servers` package only provides an abstract tagged type `Execution_Server`. The run-time is responsible for providing the server implementations by the extending this type, with the implementation defining how tasks consume the shared execution budget, when the budget is replenished and how the execution server operates when there is no task to dispatch. The `Execution_Server` object does not need to directly implement the execution server; instead, it may act as a proxy to other run-time objects.

There are two proposals for the definition of the `Execution_Server` type and how objects of this type are declared: the first proposal takes on a task-like persona but requires compiler support, while the second proposal is implementable within the existing confines of Ada. The presentation of the two options allows a discussion on the best form the `Execution_Server` type should take.

Proposal 1 — Aspect Approach

Under this proposal, the definition of the `Execution_Server` type is:

```
type Execution_Server is abstract tagged limited private;
```

Applying to object declarations of types that extend the `Execution_Server` type is a new representation aspect `Execution_Server_Object` of type `Boolean`. Using this aspect informs the user and compiler that the object is an execution server. In response, the initialisation of an execution server object now includes the creation of any run-time objects associated with the execution server and registers the server with the

run-time. The `Priority`, `Cycle_Period`, `Cycle_Phase`, `Execution_Budget` `Relative_Deadline` and `CPU` representation aspects also apply to object declarations of the `Execution_Server` type when the aspect `Execution_Server_Object` is provided: providing a means to set the timing parameter of the execution server.

For example, an implementation may provide a *Priority Server* as an execution server implementation. Under this proposal a declaration of a Priority Server object would look like:

```
My_Priority_Server : Priority_Server
  with Execution_Server_Object,
       Priority          => 16,
       Cycle_Period      => Seconds (10),
       Cycle_Phase       => Seconds (1),
       Execution_Budget  => Seconds (3),
       Relative_Deadline => Seconds (4);
```

Proposal 2 — Discriminate Approach

Under this proposal, the definition of the `Execution_Server` type includes an unknown discriminant part:

```
type Execution_Server (<>) is abstract tagged limited private;
```

Initialisation and registration with the run-time of the execution server object and any run-time components is carried out with the `New_Execution_Server` function:

```
function New_Execution_Server
  (Priority          : System.Any_Priority;
   Period            : Real_Time.Time_Span;
   Phase             : Real_Time.Time_Span;
   Execution_Budget  : Real_Time.Time_Span;
   Relative_Deadline : Real_Time.Time_Span := Real_Time.Time_Span_Last;
   CPU               : System.Multiprocessors.CPU_Range := Not_A_Specific_CPU)
  return Execution_Server is abstract;
```

The example in the first proposal would become:

```
My_Priority_Server : Priority_Server :=
  New_Execution_Server (Priority          => 16,
                        Cycle_Period      => Seconds (10),
                        Cycle_Phase       => Seconds (1),
                        Execution_Budget  => Seconds (3),
                        Relative_Deadline => Seconds (4));
```

Proposal Comments

No extra compiler support is required for the second execution server type and has a straightforward implementation. On the other hand, the first proposal requires changes to the compiler to support the application of new and existing representational aspects to object declarations of types that extend `Execution_Server`. It also requires the compiler to insert an associated run-time call to initialise the execution server object and register it with the run-time.

The advantage of the first proposal is it allows the properties of the execution server to be set in the same manner as for tasks. This view can be important, as execution servers from the point of view of the underlying scheduler are nothing more than a periodic task. Thus, the execution server in the first proposal has the look of a task, and allows external tools to read and write the cyclic properties of tasks and execution servers in the same way.

4 Examples

A simple periodic task counts the number of elapsed seconds, printing the value of the counter each cycle. Using the new syntax:

```
task Counter
  with Priority       => 15,
       Cycle_Behavior => Periodic,
       Cycle_Period   => Seconds (1);

task body Counter is
   J : Natural := 0;
begin
   Put_Line ("Counter Start");
   Put (J);
cycles
   J := J + 1;
   Put (J);
end Counter;
```

Notice how the body of the task does not contain any task timing information; instead, that information is can be found in clear and easy format on the task specification. The only cyclic information the user needs to care about in the body of the task is their cyclic code.

A user though may desire a collection of counter tasks that increment at different rates. Reusing the same body, the task declaration would become:

```
task type Counter (Rate_In_Seconds : Positive)
  with Priority       => 15,
       Cycle_Behavior => Periodic,
       Cycle_Period   => Seconds (Rate_In_Seconds);
```

And the user can now create counters with different increment rates:

```
Counter_1 : Counter (Rate_In_Seconds => 1);
Counter_2 : Counter (Rate_In_Seconds => 5);
```

Or even an array of counters:

```
Array_Of_Counters : array (1 .. 100) of Counter (Rate_In_Seconds (60));
```

In this example, a sporadic task counts the number of times a big red button is pressed, ignoring any subsequent presses for five second. Once the button has been pressed more than ten times the system will self destruct. Note this example demonstrates the use of an exception handler and how the tasking abstraction provides a scope that makes the `Button_Press_Counts` type only visible to the body of the task — the real-time utilities library would have exposed this type to the enclosing package.

```
task Red_Button_Monitor
  with Priority       => 15,
       Cycle_Behavior => Sporadic,
       Cycle_Period   => Seconds (5),


task body Red_Button_Monitor is
   type Button_Press_Counts is range 0 .. 10;
   Press_Count : Button_Press_Counts := 0;
begin
   null;
cycles
   Press_Count := Press_Count + 1;
```

```
exception
   when Contrainst_Error =>
      Self_Destruct;
end Red_Button_Monitor;
```

An interrupt handler attached to the button would inform the task that the button has been pressed through the procedure call:

```
Ada.Cyclic_Tasks.Release_Task (Red_Button_Monitor'Identity);
```

This final example revisits the cyclic task specified in Table 1 using the new syntax:

```
protected Handler_Object is
   procedure Missed_Deadline (T : Task_Id);
   procedure Budget_Exhaustion (T : Task_Id);
end Handler_Object;

protected body Handler_Object is
   procedure Missed_Deadline (T : Task_Id) is
   begin
      ... missed deadline handler ...
   end Missed_Deadline;

   procedure Budget_Exhaustion (T : Task_Id) is
   begin
      ... cycle execution overrun handler ...
   end Budget_Exhaustion;
end Handler_Object;

task type My_Cyclic_Task
  with Priority           => 15,
       Cycle_Behavior     => Periodic,
       Cycle_Period       => Seconds (1),
       Cycle_Phase        => Milliseconds (200),
       Relative_Deadline  => Milliseconds (500),
       Deadline_Response  => Handler,
       Deadline_Handler   => Handler_Object.Missed_Deadline'Access;
       Execution_Budget   => Milliseconds (100),
       Execution_Response => Handler,
       Execution_Handler  => Handler_Object.Budget_Exhaustion'Access;

task body My_Cyclic_Task is
   ... task state ...
begin
   ... initialise task ...
cycles
   ... cyclic code ...
end Cyclic_Task;

My_Task : My_Cyclic_Task;
```

5 Implementation

A subset of the cyclic task behaviour consistent with the Ravenscar Profile has been implemented within the GNAT compiler, targeting a new real-time executive called Acton. Designed to run on 32-bit microcontrollers, Acton provides a new Ada tasking run-time. The motivation behind Acton was to provide a real-time executive that would rectify issues that are present in an existing Ada real-time executive — GNAT for Bare

Boards. Unlike GNAT for Bare Boards, Acton is able to support multiple task dispatching policies; track the execution time of a task accurately; and ensure that low priority tasks and software timers do not interrupt a higher priority task.

Acton supports the dynamic cyclic task semantics natively within its kernel. This approach allows the compiler to transform the task body from the last example into:

```
task body My_Cyclic_Task is
   ... task state ...
begin
   ... initialise task ...
   Begin_Cycles_Stage;
   loop
      New_Cycle;
      ... cyclic code ...
   end loop;
end My_Cyclic_Task;
```

Where `Begin_Cycles_Stage` and `New_Cycle` are kernel calls, with the latter call blocking the task until the next task cycle can commence. The cyclic task properties for the task are provided to the kernel as part of the kernel's task create call.

For Ravenscar Profile systems, most of the cyclic task behaviour is available. Removed are the `Ada.Dynamic_Cyclic_Tasks` and `Ada.Dynamic_Execution_Servers` packages, as dynamically changing cyclic task properties introduces non-deterministic tasking behaviour. Also removed are the `Abort_Cycle` and `Raise_Exception` deadline and execution budget responses as they introduce asynchronous transfer of control into the real-time program.

6 Language Impact

Providing cyclic tasks through a library-based approach is preferred over the inclusion of a cyclic task abstraction into the language, since the latter requires potential widespread modification to the language; and in turn to the compilers and run-times that implement the language. Thus, significant change to the language quickly diminishes any appetite for language modification, more so when the changes affect the wider Ada community.

For example, Ted Baker in [3] presented an approach to incorporate simple periodic tasks within Ada. His approach was to provide the ability to specify whether an Ada task was periodic — and to give its periodicity if needed — and made use of a modified delay statement that would delay the task until its next period. The Ada 9X Requirements Team considered this proposal for inclusion into the Ada 9X Revision Requirements [4], but rejected it on the grounds:

> *"A generalized specification of 'periodic' would generate a complete new task class within Ada. Its behaviour in rendezvous, priority, and possible restrictions make this language change expensive. Since all of the required functionality can be provided with [delay until statement], it is judged out-of-scope."*

While the cyclic task semantics presented in this paper follows an approach analogous to Baker, it does not create a new task class that needs to be treated differently by other language features: for example a task with cyclic semantics behaves no differently in rendezvous than a task without. In detail, the cyclic semantic extension to the task body consists of three parts:

- A sequence of code, executed cyclically.
- A release mechanism at the start of this sequence that:

    - Blocks the task until a new task cycle can commence.
    - Sets up the timers associated with tracking the new task cycle's deadline and execution budgets.
- A mechanism to handle a task cycle's missed deadline or exceeded execution budget.

By placing the cyclic code in its own, non-nested section with an implied release mechanism, under normal conditions — that is a task does not miss its deadline or exceed its execution budget — a task cannot complete a task cycle while performing any task interaction: for example a rendezvous, a protected action or a select statement. This is because a task has to complete its current task cycle — and the statements within it — before it can commence its next, and the syntax of the cyclic section does not allow task interaction to span across task cycles. Thus, a task executing a task cycle interacts no differently with Ada's tasking features than an existing non-cyclic task.

If a task does miss its deadline or exceeds its execution budget, support for the three run-time responses — call handler, abort current task cycle and raise exception — already exist in Ada. Handlers behave in the same manner as handlers for timing events and execution time timers; aborting the current task cycle is able to use Ada's abort of a sequence of statements (ARM 9.8); and exception handling is clearly defined by the language.

Thus, the approach taken to introduce cyclic task semantics to Ada tasks confines the language changes to the task unit and any asynchronous tasking operation — for example the abort statement — that occurs while the task is blocked on the release mechanism. The design also ensures that Ada's existing tasking behaviour remains unmodified when a cyclic section is not present. However, these ramifications to the language remain:

- Increase complexity of Ada tasks;
- A new keyword.

To help reduce the impact on the core language, only the new task body syntax and the default yield cyclic semantic need to be included within the core language. The new cyclic task properties, their modification to the default semantics, deadline and execution time tracking, and executions server support would then exist within the Real-Time Annex. The requirement though to change the core language and the impact a new keyword will have on existing software means acceptance of the basic cyclic task semantics is required from the wider Ada community — a level a library-based approach does not require.

7 Conclusion

To close the gap between the design and specification of an Ada task and its implementation in Ada, this paper has presented a new implementation approach requiring an extension to the syntax and semantic of Ada's tasks. This approach — unlike existing ones — provide a natural expression of cyclic tasks in a manner that would be familiar to Ada users. The user no longer needs to consider a cyclic task as a set of components loosely linked together, making it easier to implement and understand a cyclic task expressed in Ada.

The cost of this approach is the semantics of tasks — and thus the language itself — becomes larger and more complex with the addition of the new optional cyclic semantics; the introduction of a new keyword will also affect existing code. What is lost with a more complex language is gained through the clarity and structure brought to the implementation of a cyclic task that is not currently available to users. When the user elects to use only static cyclic properties, this effect is more pronounced: all these properties are now neatly listed on the task type's declaration and the compiler ensures these properties cannot be changed at run-time.

These benefits extend themselves to the software tools that real-time users use to ensure that their software meets its specification, opening up interesting possibilities. By defining the static properties of a cyclic task in a standardised approach, it opens the door for software tools to read and write the properties straight from the code. This presents a greater opportunity for real-time support and design tools to interact with a program's

code. For example, a worst-case estimation time tool can annotate a program with its calculations for the execution time bounds for the program's tasks: then have the run-time enforce these bounds at run-time. A schedulability analysis tool now has the ability to read the same task properties as the run-time, enabling it to check the schedulability of a real-time system straight from the code. To demonstrate the viability of this direct tool–code interaction, a simple ASIS tool has been developed that lists the tasks contained with a program along with the cyclic properties of each task.

8 References

[1] A. Wellings and A. Burns, "Real-Time Utilities for Ada 2005," in *Reliable Software Technologies – Ada-Europe 2007, Proceedings*, Geneva, Switzerland, 2007, pp 1–14.

[2] G. C. Buttazzo, *Hard Real-Time Computing Systems*, 3rd ed. Boston, MA, USA: Springer, 2011.

[3] T.P. Baker, "Fixing Some Time-Related Problems in Ada", in *Third International Workshop on Real-Time Ada Issues, Proceedings*, Nemacolin Woodlands, PA, June 1989, pp 136-143.

[4] Ada 9X Project Office, "Ada 9X Project Report: Ada 9X Revision Issues Release 2," Office of the Under Secretary of Defense for Acquisition, AD-A223 166, May 1990, pp351–352.

9 Appendix

9.1 Ada.Cyclic_Tasks

```
limited with Ada.Task_Identification;
with Ada.Real_Time;
package Ada.Cyclic_Tasks is
   type Behaviour is (Yield, Aperiodic, Sporadic, Periodic);
   type Event_Response is (No_Response, Handler, Abort_Cycle,
                           Abort_And_Raise_Exception);
   type Response_Handler is
     access protected procedure (T : in Ada.Task_Identification.Task_Id);

   procedure Release_Task (T : in Ada.Task_Identification.Task_Id);

   function Get_Cycle_Period
     (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Ada.Real_Time.Time_Span;
   function Get_Cycle_Phase
     (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Ada.Real_Time.Time_Span;

   function Get_Relative_Deadline
     (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Ada.Real_Time.Time_Span;
   function Get_Absolute_Deadline
    (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Ada.Real_Time.Time;
   function Has_Deadline_Passed
    (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Boolean;

   function Get_Execution_Budget
     (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Ada.Real_Time.Time_Span;
```

```
   function Get_Remaining_Budget
     (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Ada.Real_Time.Time_Span;
   function Is_Budget_Exhausted
     (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Boolean;

   function Get_Deadline_Response
     (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Event_Response;
   function Get_Budget_Response
     (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Event_Response;

   function Get_Deadline_Handler
     (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Response_Handler;
   function Get_Budget_Handler
     (T : Ada.Task_Identification.Task_Id := Ada.Task_Identification.Current_Task)
     return Response_Handler;
end Ada.Cyclic_Tasks;
```

9.2 Ada.Cyclic_Tasks.Dynamics

```
package Ada.Cyclic_Tasks.Dynamics is
   procedure Set_Cycle_Period (Period : Ada.Real_Time.Time_Span;
                               T      : Ada.Task_Identification.Task_Id :=
                                           Ada.Task_Identification.Current_Task);
   procedure Set_Cycle_Phase (Period : Ada.Real_Time.Time_Span;
                              T     : Ada.Task_Identification.Task_Id :=
                                         Ada.Task_Identification.Current_Task);
   procedure Set_Relative_Deadline
     (Deadline : Ada.Real_Time.Time_Span;
      T        : Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);
   procedure Set_Execution_Budget
     (Budget : Ada.Real_Time.Time_Span;
      T      : Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);
   procedure Extend_Current_Budget
     (Amount : Ada.Real_Time.Time_Span;
      T      : Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);
   procedure Set_Deadline_Response
     (Response : Event_Response;
      T        : Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);
   procedure Set_Budget_Response
     (Response : Event_Response;
      T        : Ada.Task_Identification.Task_Id :=
                    Ada.Task_Identification.Current_Task);
   procedure Set_Deadline_Handler
     (Handler : Event_Response;
      T       : Ada.Task_Identification.Task_Id :=
                   Ada.Task_Identification.Current_Task);
   procedure Set_Budget_Handler
     (Handler : Event_Response;
```

```
      T       : Ada.Task_Identification.Task_Id :=
                   Ada.Task_Identification.Current_Task);
end Ada.Cyclic_Tasks.Dynamics;
```

9.3 Ada.Execution_Servers

```
with Ada.Task_Identification;
package Ada.Execution_Servers is
   type Execution_Server is abstract tagged limited private;

   function Get_Execution_Server
     (T      : Ada.Task_Identification.Task_Id :=
                  Ada.Task_Identification.Current_Task)
     return Execution_Server is abstract;
   function Is_Task_In_Server
     (T      : Ada.Task_Identification.Task_Id :=
                  Ada.Task_Identification.Current_Task;
      Server : Execution_Server)
     return Boolean is abstract;

   ... the Get_* functions from Ada.Cyclic_Tasks have their analogues here ...
   ... where applicable ...
end Ada.Execution_Servers;
```

9.4 Ada.Execution_Servers.Dynamics

```
package Ada.Execution_Servers.Dynamics is
   procedure Add_Task_To_Server
     (T      : Ada.Task_Identification.Task_Id :=
                  Ada.Task_Identification.Current_Task;
      Server : Execution_Server)
   procedure Remove_Task_From_Server
     (T      : Ada.Task_Identification.Task_Id :=
                  Ada.Task_Identification.Current_Task;
      Server : Execution_Server)

   ... the Set_* procedures from Ada.Cyclic_Tasks.Dynamics have their ...
   ... analogues here where applicable ...
end Ada.Execution_Servers.Dynamics;
```