



Deposited via The University of York.

White Rose Research Online URL for this paper:

<https://eprints.whiterose.ac.uk/id/eprint/106671/>

Version: Accepted Version

Article:

Wellings, A. J., Cholpanov, V. and Burns, A. (2016) Implementing Safety-Critical Java Missions in Ada. ACM Ada Letters. pp. 51-62. ISSN: 1094-3641

<https://doi.org/10.1145/2971571.2971578>

Reuse

Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown

If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

Implementing Safety-Critical Java Missions in Ada

A.J. Wellings, V. Cholpanov and A. Burns
Department of Computer Science, University of York, UK
andy.wellings, alan.burns@york.ac.uk

Abstract

Critical systems written in Ada are still reluctant to use dynamic memory allocation. The Ravenscar profile, for example, prohibits the dynamic creation of tasks. This is in spite of the availability of storage pools and the strong compile-time checking of access types. The Java community has, by necessity, taken a slightly less conservative approach. Safety-Critical Java (SCJ) supports a constrained use of dynamic memory allocation. This paper takes the SCJ approach and tries to implement it using Ada's storage pools. We show that the approach is not directly transferable to Ada due to the difference in the way that SCJ and Ada handle region-based memory management. However, an equivalent approach can be developed.

1 Introduction

The fabric of society is now almost completely dependant on the correct operations of computer systems. Such systems are called *critical systems*, and their failure can result in threat to human life, environmental harm or significant economic losses. It has become increasingly common to classify such systems as being safety-critical, mission critical or business-critical. For critical systems, the most important property is their dependability. This reflects the users' degree of trust in their correct operation. Critical systems were once small and very conservative programming methods were used during their construction. For example, applications were single threaded and did not dynamically allocate objects (preferring to preallocate their objects during an initialization phase and then reuse the space allocated if necessary). Nowadays, critical system have become much more complex. Multiple-threaded applications are more typical and dynamic behaviour is increasingly expected.

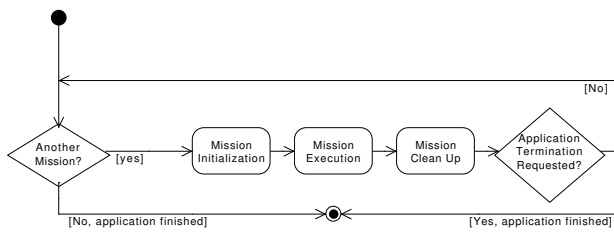
The Ada community has responded to this shift in expectations by focusing on the definition of a subset of the language; the use of which results in predictable program execution. Here, we use the term predictable to mean that the worst-case response times of tasks can be determined, along with their maximum memory demands. For safety-critical application, dynamic memory allocation is still prohibited, and a very static tasking profile is required.

The Java community has taken a slightly less conservative approach. The Java language is intrinsically dynamic, almost exclusively relying on dynamic object creation. Any approach that prohibits dynamic memory allocation is considered an anathema. However, the community also recognises that any methodology that requires garbage collection is unlikely to be well received by the authorities that certify that critical systems are dependable. Safety-Critical Java (SCJ – a subset of Java augmented by the Real-Time Specification for Java) supports a novel application structure based on the notion of *missions*. An application's lifetime is defined by the execution by a *mission sequencer* of a *sequence* of missions as illustrated in Figure 1.(a).

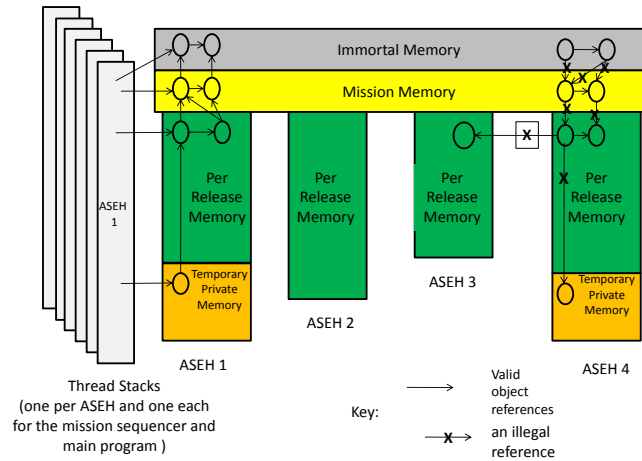
Each mission is comprised of a set of real-time concurrent activities called asynchronous event handlers (ASEH¹) that are created during the mission's initialization phase, and started when the mission enters its execution phase. Each ASEH executes a sequence of jobs (called releases), and is classified as being either periodic or aperiodic. When all the ASEHs have terminated, a clean-up phase is performed and the next mission (if any) is initialized and executed. The application can request that a mission (or the whole application) be terminated. When mission termination is requested, all ASEHs are allowed to finish their current releases (job) before the mission is terminated. When application termination is requested, the current mission is requested to terminate first. When that terminates, the application terminates.

Dynamic memory allocation is allowed but is restricted so that it follows a well-defined pattern of usage. Each application, effectively, has a fixed block of memory allocated to it, which is called the *immortal memory area*. This memory is intended

¹In fact, the SCJ supports both event handlers and real-time threads. For simplicity, we shall just focus on the SCJ Level 1 programming model, which only allows ASEH. However, we will use Ada tasks as the executor of concurrent activities.



(a) The Mission Life Cycle Diagram



(b) The SCJ Memory Model

Figure 1: The SCJ Model

for dynamically-created objects that once created will remain for the lifetime of the application. The SCJ virtual machines determines the size of this memory but, usually, this is a configuration parameter.

Each mission also has a fixed block of memory allocated to it, which is called the *mission memory area*. The size of this block is set by the application. Objects can be allocated in this area, but when the mission is finished all the memory is reclaimed and the block reused by the subsequent mission. Each real-time activity, also has access to its own block of memory, called its private memory (again the size is set by the application). Objects allocated here can *only* be accessed by the owing ASEH. The private memory is cleared at the end of each release of the ASEH. Hence, the term per-release memory area is often used to identify this area. Activities can also have nested private memory areas that can be used to store objects that have an even more limited lifetime (perhaps for the lifetime of a method call). Static analysis or run-time checks ensure memory safety, by which we mean that no dangling pointers can occur. Figure 1.(b) illustrates the memory structure of an SCJ program. In addition to the above memory areas, each ASEH also has a thread stack where local variables declared in methods reside. The figure shows four ASEHs and a mission sequencer and main program threads of execution.

The term *backing store* is used by the SCJ to indicate the block of memory that provides the space where objects in a *memory area* are stored. The term *memory area* is used to indicate the backing store and any control variables needed for its management.

The goal of this paper is to discuss the feasibility of implementing the SCJ mission concept in Ada. This would allow more dynamic behaviour for programs than currently can be obtained by a Ravenscar-compliant safety critical Ada program. However, the approach still imposes a restricted use of dynamic memory management that is free of fragmentation problems and out of memory exceptions. Our aim is to use Ada's storage pools in combination with the scope rules for access types to represent the immortal, mission, per release and nested private memory. The challenge is to ensure that memory safety of the Ada program is not violated.

Section 2 briefly reviews Ada's support for dynamic memory, and discusses how memory pools can be used. We then focus our discussions on a Level One application consisting of just periodic tasks. Section 3 discusses how a standard representation of a periodic task can be updated to support per-release memory, with no nested private memories. Section 4 then considers how mission memory can be implemented. Section 5 provides a prototype API and implementation of an Ada-based Level 1 system; this is followed in Section 6 by a simple example of how it can be used. Related work is briefly considered in Section 7 and Section 8 provides our conclusions.

2 Dynamic Memory Management in Ada

Ada memory can be allocated dynamically either on a task's stack or via an allocator. In both cases, references to the created objects can be created. Here, we focus on objects created by an allocator.

The space for objects created by an allocator, by default, come from a standard storage pool (heap) for each access-to-object type. There is no explicit garbage collector defined; the default storage pool need not support deallocation of individual objects. However, the scope rules of the language define when objects can no longer be referenced (this is when the associated access types goes out of scope) and, therefore, any objects created using that access types are eligible for collection. *The language, however, does not requires that they be collected* .

“By default, the implementation might choose to have a single global storage pool, which is used (by default) by all access types, which might mean that storage is reclaimed automatically only upon partition completion. Alternatively, it might choose to create a new pool at each accessibility level, which might mean that storage is reclaimed for an access type when leaving the appropriate scope. Other schemes are possible.”

A program can define its own storage pool using the `System.Storage_Pools` package. This allows an application to define its own memory management. For example, the following package specification illustrates an application-defined storage pool

```
with System, System.Storage_Pools, System.Storage_Elements;
use System, System.Storage_Pools, System.Storage_Elements;
package Custom_Pools is
  type Custom_Pool (Pool_Size : Storage_Count) is new Root_Storage_Pool with private;
  type Custom_Pool_Access is access all Custom_Pool;
  overriding procedure Allocate(Pool : in out Custom_Pool; Storage_Address : out Address;
    Size_In_Storage_Elements : in Storage_Count; Alignment : in Storage_Count);
  overriding procedure Deallocate(Pool : in out Custom_Pool; Storage_Address : in Address;
    Size_In_Storage_Elements : in Storage_Count; Alignment : in Storage_Count);
  overriding function Storage_Size(Pool : Custom_Pool) return Storage_Elements.Storage_Count;
  overriding procedure Initialize(Object : in out Custom_Pool);
  overriding procedure Finalize(Object : in out Custom_Pool);
private
  type Custom_Pool (Pool_Size : Storage_Count) is new Root_Storage_Pool with
    record
      Storage : Storage_Array (1 .. Pool_Size);
      ...
    end record;
end Custom_Pools;
```

Note that here the pool itself is stored within the extended type. Hence, its space will be allocated wherever the space for an object of the `Custom_Pool` type is allocated. Note, also that as an extension of a `Limited_Controlled` type, initialization and finalization routines can be provided. Further note, `Deallocate` is *only* called when the program uses unchecked deallocations.

In order to use a pool, it is necessary to associate an access type with the pool. For example:

```
My_Pool : Custom_Pool(1000);
type My_Integer is access Integer;
for My_Integer'Storage_Pool use My_Pool;
```

More information on the use of Ada's storage pools can be found on the AdaCore website (http://www.adacore.com/uploads_gems/07_safe_secure_ada_2005_safe_memory_management.pdf).

3 Per-Release Memory

In the SCJ, a periodic activity is assigned its own private memory area in which it can allocate objects whose lifetime is only for a single release; data that persists between releases is stored in mission memory. There are several ways that this behaviour could be achieved in Ada. Consider a simple template for a periodic task. Here, we present only the basic structure to illustrate the approach. So, for example, we are assume deadline equals period, no deadline miss detection, no initialization etc.

```
task type Periodic(Period_In_Milliseconds : Positive);
task body Periodic is
  Next_Release : Time := Clock;
begin
  loop
    -- code
    Next_Release := Next_Release + Milliseconds(Period_In_Milliseconds);
    delay until Next_Release;
  end loop;
end Periodic;
```

This structured is easily amended to support private per task dynamic memory. Suppose, for example the task wanted to create a dynamic array that would be freed at the end of each release. As shown below, `Array_Size` is an added discriminant to the task type to illustrate the amended structure.

```
task type Periodic(Period_In_Milliseconds : Positive; Array_Size : Positive);
```

We now consider the various ways in which this per-release memory areas can be achieved.

3.1 Heap-allocated objects

The first approach simply allocates memory from the heap and uses the Ada scope rules to control its lifetime.

```
task body Periodic is
  Next_Release : Time := Clock;
  -- Objects allocated from access types declared here will be private to the task but
  -- persist between releases, e.g.
  type Accumulated_Integer is access Integer;
  Accumulator : Accumulated_Integer := new Integer;
begin
  Accumulator.all := 0;
  loop
    declare
      -- Objects allocated from access types declared here will be local to the task
      -- and can be collected between releases, e.g.
      type Local_Integers is array(1..Array_Size) of Integer;
      type Local_Access is access Local_Integers;
      Local : Local_Access := new Local_Integers;
      Local_Sum : Integer;
    begin
      -- dynamically fill in array, perhaps from a device, and produce Local_Sum
      Accumulator.all := Accumulator.all + Local_Sum;
    end;
    Next_Release := Next_Release + Milliseconds(Period_In_Milliseconds);
    delay until Next_Release;
  end loop;
end Periodic;
```

The advantage of this code is that it is type safe, i.e. there is no use of unchecked accesses – as long as the task body does not try to pass references to the data to any external library package. If it needed to do this then unchecked access will be needed, as the array is not global.

The main disadvantage of the approach is that it is not possible to control the memory allocation explicitly. The programmer must rely on the implementation to collect when access types go out of scope and hope that fragmentation of the heap does not occur.

3.2 Stack-allocated objects

Using the scope rules of Ada, it is possible to achieve the same effect as that illustrated in Section 3.1 without dynamic memory allocation by declaring objects on the stack. For example:

```
task body Periodic is
  Next_Release : Time := Clock;
  -- Objects allocated declared here persist between releases, e.g.
  Accumulator : Integer;
begin
  Accumulator := 0;
  loop
    declare
      -- Objects declared here will be local to the task
      -- and collected between releases, e.g.
      type Local_Integers is array(1..Array_Size) of Integer;
      Local : Local_Integers; Local_Sum : Integer;
    begin
      -- dynamically fill in array, perhaps from a device, and produce Local_Sum
      Accumulator := Accumulator + Local_Sum;
    end;
    Next_Release := Next_Release + Milliseconds(Period_In_Milliseconds);
    delay until Next_Release;
  end loop;
end Periodic;
```

The solution is type safe, and now there is no memory allocation from the heap during each release. If the application needs to use pointers, the objects can be aliased. All data is allocated on the task's stack. The `Storage_Size` attribute must be used to specifies the amount of space to be allocated for the task stack².

```
task type Periodic(Period_In_Milliseconds : Positive; Array_Size : Positive; Stack_Size : Positive)
with Storage_Size => Stack_Size;
```

The disadvantage of this approach is that, once set, the stack size cannot be extended, and if the stack is exhausted, then `Storage_Error` will be raised (if stack checking is enabled). Hence, all tasks must be allocated the maximum amount of memory that they will need throughout *all* releases. This, clearly, will be pessimistic as not all tasks will need their maximum stack size at the same time.

3.3 Pool allocated objects

Using the `Custom_Pool` type defined in Section 2, we can easily extend an Ada periodic task to use this facility.

```
task body Periodic is
  Next_Release : Time := Clock;
  -- Objects allocated from access types declared will be private to the task but
  -- persist between releases, e.g.
  Accumulator : Integer;
begin
  Accumulator := 0;
  loop
    declare
      Per_Release_Memory : aliased Custom_Pool(Per_Release_Size);
      -- Objects allocate from access types declared here will be local to the task
      -- and collected between releases, e.g.
      type Local_Integers is array(1..100) of Integer;
      type Pooled_Integers is access Local_Integers;
      for Pooled_Integers'Storage_Pool use Per_Release_Memory;
      Local_Ints : Pooled_Integers := new Local_Integers;
      Local_Sum : Integer := 0;
    begin
      -- dynamically fill in array, perhaps from a device, and produce Local_Sum
      Accumulator := Accumulator + Local_Sum;
    end;
    Next_Release := Next_Release + Milliseconds(Period_In_Milliseconds);
    delay until Next_Release;
  end loop;
end Periodic;
```

However, with this simple use of pools, the space for object allocation still occurs on the stack, and hence the worst case stack usage is the sum of the maximum stack space needed for each task. If the `Custom_Pool`, is created with an allocator, then again the Ada run-time will allocate it from the heap with the problem of potential fragmentation.

However, the approach adopted by the custom pool implementation was to store the space for object allocations as part of the type. We do not need to do this (see Section 4).

4 Mission Memory

So far we have only considered per release memory and how objects can persist between releases. We haven't addressed the issue of where objects that might be shared between tasks are stored. In the SCJ, Mission Memory is used for both purposes. Ada allows us to separate out these two requirements as the task is a visible construct, and data can be explicitly placed on the stack. In Java/SCJ data can only be placed local to a method or local to the object.

In the SCJ model, objects allocated in mission memory must also be freed at the end of the mission. Hence, they cannot be declared global to an Ada package body. Such variables are equivalent to immortal variables in SCJ. However, a storage pool *can* be declared globally in a package and the space managed so that it is freed at the end of the mission. In the SCJ, the backing store that is allocated to tasks encapsulated in a mission comes from the total backing store that is assigned to that mission. Consider the following package specification that provides the equivalent to SCJ Level 1 mission memory.

²Note, Ada does not provide any API to help the programmer measure the stack size consumed by a task. However, AdaCore provide GNAT.Task_Stack_Usage.Get_Current_Task_Usage.Value for this purpose.

```

with System, System.Storage_Pools, System.Storage_Elements;
use System, System.Storage_Pools, System.Storage_Elements;
package Mission_Memory is
  -- This package manages the backing store for a mission. In a Level 1 system,
  -- there can only ever be one Mission active at once.
  type Backing_Store (Pool_Size : Storage_Count) is new Root_Storage_Pool with private;
  type Backing_Store_Access is access all Backing_Store;
  overriding procedure Allocate( Pool : in out Backing_Store; Storage_Address : out Address;
    Size_In_Storage_Elements : in Storage_Count; Alignment : in Storage_Count);
  overriding procedure Deallocate( Pool : in out Backing_Store; Storage_Address : in Address;
    Size_In_Storage_Elements : in Storage_Count; Alignment : in Storage_Count);
  overriding function Storage_Size(Pool : Backing_Store) return Storage_Count;

  overriding procedure Initialize(Pool: in out Backing_Store);
  overriding procedure Finalize(Pool: in out Backing_Store);
  procedure Reset(Pool : in out Backing_Store);
  procedure Set_Mission_Memory(Pool : in out Backing_Store);
private
  type Backing_Store (Pool_Size : Storage_Count) is new Root_Storage_Pool with
    record
      Pool_Start : Storage_Count := 1;
      Next_Allocation : Storage_Count := 1;
      Used : Storage_Count := 0;
      Is_Mission_Memory : boolean := False;
    end record;
end Mission_Memory;

```

The total backing store for a mission is declared in the package body. Its size can be set from a command line option or a configuration package. Here we use the latter approach. The mission memory's backing store will come from this global backing store, as will each mission's task private memory backing store. In Ada, the backing store is a storage pool. The package provides overridden subprograms for both the primitive operations of `Storage_Pools` and the `Limited_Controller` type which `Storage_Pools` extends. The full body of the package is given below:

```

with Configuration_Parameters, System.Address_Image, Ada.Task_Attributes, Simple_Locks;
use Simple_Locks;
package body Mission_Memory is
  Backing_Storage : Storage_Array (1 .. Configuration_Parameters.Backing_Store_Size);
  type Task_Store is record
    Across_Release : Storage_Count := 0; Per_Release : Storage_Count := 0;
  end record;
  package Pool_Position is new Ada.Task_Attributes(Task_Store, (0,0));

  protected Allocate_Controller is
    procedure Allocate_BS(Size : Storage_Count; Pos : out Storage_Count);
    function Get_Next_Pool_Start return Storage_Count;
  private
    Next_Pool_Start : Storage_Count := 1;
  end Allocate_Controller;

  protected body Allocate_Controller is separate;

  Lock : Simple_Lock; -- use to lock mission memory

  overriding procedure Allocate( Pool : in out Backing_Store; Storage_Address : out Address;
    Size_In_Storage_Elements : in Storage_Count; Alignment : in Storage_Count) is
  begin
    if Pool.Is_Mission_Memory then
      Lock.Acquire; -- mission memory allocation needs synchronization
    end if;
    Pool.Next_Allocation := Pool.Next_Allocation + ((-Pool.Next_Allocation-1) mod Alignment);
    if Pool.Used + Size_In_Storage_Elements-1 > Pool.Storage_Size then
      raise Storage_Error;
    end if;
    Storage_Address := Backing_Storage (Pool.Next_Allocation)'Address;
    Pool.Next_Allocation := Pool.Next_Allocation + Size_In_Storage_Elements;
    Pool.Used := Pool.Used + Size_In_Storage_Elements;
    if Pool.Is_Mission_Memory then Lock.Release; end if;
  end Allocate;

  overriding procedure Deallocate(Pool : in out Backing_Store; Storage_Address : in Address;
    Size_In_Storage_Elements : in Storage_Count; Alignment : in Storage_Count) is
  begin
    Pool.Next_Allocation := Pool.Next_Allocation - ((-Pool.Next_Allocation) mod Alignment);

```

```

    Pool.Next_Allocation := Pool.Next_Allocation - Size_In_Storage_Elements;
    if Pool.Next_Allocation - Size_In_Storage_Elements-1 < 1 then
        raise Storage_Error;
    end if;
    Pool.Used := Pool.Used - Size_In_Storage_Elements;
end Deallocate;

overriding function Storage_Size(Pool : Backing_Store)
    return Storage_Elements.Storage_Count is
begin return Pool.Pool_Size; end ;

overriding procedure Initialize(Pool: in out Backing_Store) is
begin
    if Pool_Position.Value.Across_Release = 0 then
        -- first allocation
        Allocate_Controller.Allocate_BS(Pool.Pool_Size, Pool.Next_Allocation);
        Pool_Position.Set_Value((Pool.Next_Allocation,0));
        if Allocate_Controller.Get_Next_Pool_Start > Configuration_Parameters.Backing_Store_Size then
            raise Storage_Error;
        end if;
    elsif Pool_Position.Value.Per_Release = 0 then
        Allocate_Controller.Allocate_BS(Pool.Pool_Size, Pool.Next_Allocation);
        Pool_Position.Set_Value((Pool_Position.Value.Across_Release,Pool.Next_Allocation));
        if Allocate_Controller.Get_Next_Pool_Start > Configuration_Parameters.Backing_Store_Size then
            raise Storage_Error;
        end if;
    else Pool.Next_Allocation := Pool_Position.Value.Across_Release; end if;
end Initialize;

overriding procedure Finalize(Pool: in out Backing_Store) is
begin null; end Finalize;

procedure Reset(Pool : in out Backing_Store) is
begin
    Pool.Pool_Start := 1; Pool.Next_Allocation := 1; Pool.Used := 0;
end Reset;

procedure Set_Mission_Memory(Pool : in out Backing_Store) is
begin Pool.Is_Mission_Memory := True; end Set_Mission_Memory;
end Mission_Memory;

```

When an object of type `Backing_Store` is created, the Ada run-time automatically calls the `Initialize` subprogram. This uses a task's attribute to keep track of the position in the global array where the calling task has been allocated backing store. In SCJ, the mission sequencer is a task also, and it is its backing store that becomes the mission memory's backing store. The task attribute is initialised to 0 indicating that no backing store has been allocated. In the SCJ paradigm, the mission sequencer is the first task to run and it is responsible for executing a mission's initialization phase. It is only during the initialization phase that application tasks can be created. A simple allocation of backing store from the global array is sufficient. Here we assume no nested private memory areas. If the task has already been allocated backing store then that backing store is reused. Note that only mission memory can have concurrent calls to allocate objects.

When tasks request allocation from a memory area (be it the mission memory area or the pre-release memory area) the appropriate pool is used and the Ada run-time automatically calls the associated `Allocate` subprogram, which allocates the space from its assigned backing store.

5 The Ada API for SCJ Level 1 Equivalence

In this section, we present a prototype API that supports the concept of SCJ Level 1 missions. For convenience, we provide a single package, whose specification includes the following declaration of the `Mission` type:

```

type Mission(Number_Of_Tasks : Positive; BS : Backing_Store_Access) is abstract
    new Limited_Controlled with private;
procedure Request_Termination(M : in out Mission'Class);
type Any_Mission is access all Mission'Class;

```

As with SCJ, a mission has three phases: an initialization phase, an execution phase and a cleanup phase. The execution phase consists of the execution of the tasks that were created during the initialization phase. A task can request that a mission should be terminated by calling the `Request_Termination` subprogram. Note that the mission must define statically the

number of tasks it will create and the mission memory those tasks can use (this latter value will be given to the application via the `Mission_Sequencer`).

The implementation of this type contains a protected object that coordinates the operation of the mission.

```
protected type Mission_Controller(Number_Of_Tasks : Positive; BS : Backing_Store_Access) is
  entry Start_Barrier(Mission_Memory : out Backing_Store_Access);
  entry Finish_Barrier;
  procedure Request_Termination;
  function Is_Termination_Requested return Boolean ;
private
  All_Ready : Boolean := False;
  All_Finished : Boolean := False;
  Terminate_Requested : Boolean := False;
end;
type Controlling_Mission is access all Mission_Controller;

type Mission(Number_Of_Tasks : Positive; BS : Backing_Store_Access) is abstract
  new Limited_Controlled with record
    MC : Mission_Controller(Number_Of_Tasks, BS);
  end record;
```

All created tasks must wait for each other to be initialised before given the go ahead to start their execution. Similarly, they must wait for each other before terminating. This behaviour is enforced by the `Schedulable` template for the task.

A `Mission_Sequencer` is responsible for executing the sequence of missions. Its type is given below

```
type Mission_Sequencer is abstract tagged limited private;
function Go_Next_Mission(MS : in out Mission_Sequencer; Mission_Memory : Backing_Store_Access)
  return Boolean is abstract;
type Any_Mission_Sequencer is access all Mission_Sequencer'Class;
```

Here we diverge from the SCJ API for reasons that will be explained in Section 5.1. Instead of the SCJ method `Get_Next_Mision` we have the function `Go_Next_Mission`. The implementation of the type contains a task that will coordinate the sequence execution.

```
task type Sequencer(MS : access Mission_Sequencer'Class);
type Mission_Sequencer is abstract tagged limited record
  S : Sequencer(Mission_Sequencer'Access);
end record;
```

Finally, the template for tasks can be given. Here we just show a simple periodic task template.

```
type Schedulable(The_Mission : access Mission'Class; Per_Release_Memory_Size : Storage_Count;
  Period_In_Milliseconds : Positive) is abstract tagged limited private;
procedure Run( S : in out Schedulable;
  Per_Release_Memory : Backing_Store_Access;
  Mission_Memory : Backing_Store_Access) is abstract;
type Any_Schedulable is access all Schedulable'Class;
```

Whose implementation is

```
task type Periodic(S : Any_Schedulable);
type Schedulable(The_Mission : access Mission'Class; Per_Release_Memory_Size : Storage_Count;
  Period_In_Milliseconds : Positive) is abstract tagged limited
  record P : Periodic(Schedulable'Unchecked_Access); end record;
```

5.1 Implementing Missions

The body of `Missions` package illustrates how missions can be implemented and the memory managed. The body of the `Mission_Controller` protected type is straightforward, and not given here. The `Request_Termination` method simply forwards the request to the `Mission_Controller`.

The task templates is:

```
task body Periodic is
  Next_Release : Time;
  My_Mission_Backing_Store : Backing_Store_Access;
begin
  S.The_Mission.MC.Start_Barrier(My_Mission_Backing_Store);
  Next_Release := Clock;
  loop
    exit when S.The_Mission.MC.Is_Termination_Requested;
```

```

    declare
        Per_Release_Memory : aliased Backing_Store(S.Per_Release_Memory_Size);
    begin
        S.Run(Per_Release_Memory'Unchecked_Access, My_Mission_Backing_Store);
    end;
    Next_Release := Next_Release + Milliseconds(S.Period_In_Milliseconds);
    delay until Next_Release;
end loop;
S.The_Mission.MC.Finish_Barrier;
end Periodic;

```

Finally, the Mission_Sequencer

```

task body Sequencer is
    BS : aliased Backing_Store(1000); -- Mission Memory
    Next_Mission : Boolean := True;
begin
    BS.Set_Mission_Memory;
    while Next_Mission loop
        Next_Mission := MS.Go_Next_Mission(BS'Unchecked_Access);
        BS.Reset; -- reset mission memory
    end loop;
end Sequencer;

```

6 Using the Mission Package

To use the Missions package it is necessary to extend the types. Here we consider a simple application consisting of two missions.

```

type My_Mission1 is new Mission with private;
overriding procedure Initialize(M : in out My_Mission1);
overriding procedure Finalize(M : in out My_Mission1);

type My_Mission2 is new My_Mission1 with private;

type My_Mission_Sequencer is new Mission_Sequencer with private;
overriding function Go_Next_Mission(MS : in out My_Mission_Sequencer;
    Mission_Memory : Backing_Store_Access) return Boolean;

type My_Periodic(The_Mission : access My_Mission1; Per_Release_Memory_Size : Storage_Count;
    Period_In_Milliseconds : Positive) is new Schedulable with private;
overriding procedure Run( S : in out My_Periodic;
    Per_Release_Memory : Backing_Store_Access;
    Mission_Memory : Backing_Store_Access);

```

Of course whether the types are private or visible is up to the application. Here we will simply have a mission consisting of two periodic tasks that have access to a shared integer. The tasks and the shared integer are to be created in mission memory and hence they should be declared within the type.

```

protected type Shared_Integer(Init : Integer) is
    function Read return Integer ;
    procedure Write(I : Integer);
    procedure Increment;
private Shared : Integer := Init;
end Shared_Integer;

type My_Periodic(The_Mission : access My_Mission1;Per_Release_Memory_Size : Storage_Count;
    Period_In_Milliseconds : Positive) is
    new Schedulable(The_Mission, Per_Release_Memory_Size, Period_In_Milliseconds ) with
    record I : Integer; end record;

type My_Mission1 is new Mission with record
    -- everything here is in mission memory
    Shared : Shared_Integer(1);
    P1 : My_Periodic(My_Mission1'Access, 20, 100);
    P2 : My_Periodic(My_Mission1'Access, 30, 400);
end record;
type My_Mission2 is new My_Mission1 with null record;

type My_Mission_Sequencer is new Mission_Sequencer with record
    First : Boolean := True;
end record;

```

Note this is different from SCJ where the initialise method creates the tasks. This cannot be done with the Ada model because of the way storage pools work. Consider a possible specification of Initialize which has the mission memory passed as a parameter.

```
procedure Initialize(M : in out My_Mission; Mission_Memory : Backing_Store_Access);
```

To place a Schedulable in Mission_Memory requires the access type to name the associated storage pool.

```
procedure Initialize(M : in out My_Mission; Mission_Memory : Backing_Store_Access) is
  type My_P is access all My_Periodic;
  for My_P'Storage_Pool use Mission_Memory.all;
  P1 : My_P;
begin P1 := new My_Periodic(M'Access, 1000, 1000); end;
```

Any task created in this way now must terminate before the Initialize subprogram returns! The Mission_Memory can only be passed at run-time. A similar argument holds for any type. Once Initialize returns the access type goes out of scope. Hence, all objects that need to be stored in mission memory must be declared within the extended Mission type. The Initialize routine can only read or update the objects;

Illustrative bodies of the above types are given below (Shared_Integer is omitted as it is straight forward).

```
overriding function Go_Next_Mission(MS : in out My_Mission_Sequencer;
  Mission_Memory : Backing_Store_Access) return Boolean is
begin
  if MS.First then
    MS.First := False;
    declare
      type My_Mission_Access1 is access My_Mission1;
      for My_Mission_Access1'Storage_Pool use Mission_Memory.all;
      M : My_Mission_Access1;
    begin
      M := new My_Mission1(2, Mission_Memory); return True;
    end;
  else
    declare
      type My_Mission_Access2 is access My_Mission2;
      for My_Mission_Access2'Storage_Pool use Mission_Memory.all;
      M : My_Mission_Access2;
    begin
      M := new My_Mission2(2, Mission_Memory); return False;
    end;
  end if;
end Go_Next_Mission;

overriding procedure Run( S : in out My_Periodic;
  Per_Release_Memory : Backing_Store_Access;
  Mission_Memory : Backing_Store_Access) is
  -- local access types for any object created in Per_Release_Memory
begin
  if S.The_Mission.Shared.Read = 10 then S.The_Mission.Request_Termination; end if;
  S.The_Mission.Shared.Increment;
end;

overriding procedure Initialize(M : in out My_Mission1) is begin M.Shared.Write(1); end Initialize;
overriding procedure Finalize(M : in out My_Mission1) is
begin
  Put_Line("Shared_at_the_end_of_the_mission_is_" & Integer'Image(M.Shared.Read));
end Finalize;
```

The implementation of the mission sequencer causes problems. The SCJ requires a Get_Next_Mission which ideally could be implemented as below:

```
function Get_Next_Mission(MS : in out My_Mission_Sequencer;
  Mission_Memory : Backing_Store_Access) return Any_Mission is
  type My_Mission_Access is access My_Mission;
  for My_Mission_Access'Storage_Pool use Mission_Memory.all;
  M : My_Mission_Access := new My_Mission(2, Mission_Memory);
begin return M; end;
```

However, this has two difficulties. First, as the Initialize method, the tasks declared must terminate before function can return. Also, it is not possible to get the return type to match the Any_Mission type, as a new type must be introduced in order to use the storage pool. One way around this is to convert using M.all.UnrestrictedAccess.

For this reason, we modify the API and provide the Go_Next_Mission function. The Go_Next_Mission is responsible for creating the mission. The return value indicates whether there is another mission to be run.

7 Related Work

Memory management for real-time systems is a well researched topic. Both the issue of predictable memory management (e.g. [3]) and real-time garbage collection (e.g. [1]) have been extensively covered. Region-based memory management [5] has provided a coarse grain deallocation mechanism [2] that increases the predictability of memory deallocation at the cost of a more complicated program design. The RTSJ scoped memory areas (on which the SCJ's memory areas are based) is an example of region-based memory allocation [4].

Ada's approach to dynamic memory allocation and deallocation via an allocator does, in essence, follow a region-based memory management approach. The scope of an access type defines the region of computation over which the lifetime of objects created from that type exists. There is a stark contrast between this approach and that of scoped memory areas. In the RTSJ/SCJ there is the notion of a *current allocation* context. When an object is created, the memory allocation will occur from the currently active context. In Ada, the memory allocated via an access type is tied to a memory pool (be it the default pool or an application-defined pool). The approach is more memory safe as assignments between access types can be fully checked by the compiler. In the RTSJ/SCJ there is the need for static analysis tools such as the Veriflux from aicas (see <https://www.aicas.com/cms/en/veriflux>). However, there has been work on *ownership types* which allows for more type security at the cost flexibility [6].

8 Conclusions

This work has evaluated the possibility of implementing the SCJ mission model in Ada. Our goal was simply to replace the SCJ allocation context of mission and per release memory area with Ada's storage pools. We attempted to do a simple translation between the SCJ API and the Ada API by mapping classes in SCJ to tagged types in Ada.

During the translation it became clear that there was a significant difference between the Ada and the SCJ approaches to region-based memory management that made the simple translation not viable. The core of the problem is that space for an object allocation in SCJ depends on the current allocation context of the creator not on any property of the class. Hence two different objects from the same class definition can be created in different memory areas. This cannot occur in Ada as the storage pool of an access variable determines where the object will be created. In order to dynamically create two objects from the same type in two different pools, two access types must be used. These will not be the same type.

A restriction of the Ada model is that it is not possible to manage the stack space of a task directly.

In terms of creating a profile that might operate with this mission model, a restriction that indicated that *only access types with associated storage pools* would be needed. Furthermore, ideally one would like for restrictions to be applicable at a finer granularity than a whole partition. This would facilitate the construction of a trusted infrastructure that has more permissions than the application.

References

- [1] David F Bacon, Perry Cheng, and VT Rajan. A real-time garbage collector with low overhead and consistent utilization. In *ACM SIGPLAN Notices*, volume 38, pages 285–298. ACM, 2003.
- [2] A. Borg, A. Wellings, C. Gill, and R.K. Cytron. Real-time memory management: life and times. In *Real-Time Systems, 2006. 18th Euromicro Conference on*, pages 11 pp.–250, 2006.
- [3] Miguel Masmano, Ismael Ripoll, Alfons Crespo, and Jorge Real. Tlsf: A new dynamic memory allocator for real-time systems. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 79–88. IEEE, 2004.
- [4] F. Pizlo, J. M. Fox, D. Holmes, and J. Vitek. Real-time java scoped memory: Design patterns and semantics. *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*, 0:101–110, 2004.
- [5] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997.
- [6] Tian Zhao, Jason Baker, James Hunt, James Noble, and Jan Vitek. Implicit ownership types for memory management. *Science of Computer Programming*, 71(3):213 – 241, 2008.

Summary of Issues to be Discussed at Workshop

This paper has identified several issues that we feel would be worth discussing at the Workshop

- There is no requirement on an implementation to cleanup the memory when an access types goes out of scope.

“By default, the implementation might choose to have a single global storage pool, which is used (by default) by all access types, which might mean that storage is reclaimed automatically only upon partition completion. Alternatively, it might choose to create a new pool at each accessibility level, which might mean that storage is reclaimed for an access type when leaving the appropriate scope. Other schemes are possible.”

You can use `Unchecked_Deallocation` but this requires the program to keep track of all objects created.

Should the language provide optional collection that can be turned off? In the same way that you can turn off null pointer checks.

- There is no way to control the space allocated for task stacks, and consequently no way that it can be integrated with storage pools. Hence there is no way for the program to ensure that fragmentation of the “stack area” does not occur. The program can set the stack size but not its location. Note there isn’t anyway specified in the RM to monitor task stack size. Although AdaCore provides an API.

Should the language (Systems Programming Annex) provide more control packages for stack manipulation?

- There is no restriction that allows constrained dynamic memory allocation. E.g “access types only to storage pools or stack objects”. Should there be?
- Restrictions apply at a partition level. It is not possible to support a “trusted” set of libraries and then application code that uses a different language subset. Should there be such a mechanism?