

# Using the Ravenscar Profile for Space Applications: the OBOSS Case

Tullio Vardanega  
European Space Research and Technology Centre  
Keplerlaan 1, 2200 AG Noordwijk, NL  
tullio@ws.estec.esa.nl

Gert Caspersen  
TERMA Elektronik  
Bregnerodvej 144, 3460 Birkerød, DK  
gec@terma.com

## Abstract

*The Ravenscar Profile emerged in 1997 as a most promising approach to bringing mature, predictable and efficient concurrency into real-time systems with integrity requirements. One of the crucial questions that have arisen since is whether the expressive power of the profile would be truly sufficient for the effective construction of such systems. This paper provides positive evidence to this effect by discussing the merit of the Ravenscar compliant architecture of a reuse library that supports the construction of on-board embedded real-time systems of the new generation. This paper argues that the notions of the profile especially contributed to increase the maturity of the reuse product. As true object orientation is the next frontier of the product, this paper also reflects on the importance of integrating the profile with suitable flavours of object orientation.*

## 1. Introduction

The Ravenscar Profile [1] defines a tasking profile that is amenable to static analysis and is especially suitable for the direct expression of concurrency in high-integrity applications [12]. [11] contended that mapping rules that conform to the Ravenscar Profile may be usefully specified for the active terminal objects defined by the HRT-HOOD design method [3], arguably one of the best equipped methods for the design and implementation of real-time systems. One of the questions that immediately arise from this argument however, is whether the expressive power of the Ravenscar Profile, dramatically inferior to that provided by the base language, suffices for the construction of real-life real-time systems. This question is significant and this paper addresses it on the basis of a practical exercise recently performed at the European Space Research and Technology Centre (ESTEC).

An ESTEC project carried out in 1996 produced a library of reusable software components designed for the construction of data handling systems compliant with the

Packet Utilisation Standard (PUS) [6] for use on board new-generation satellite systems. Data handling systems of this kind represent a crucial component of the spacecraft, for they cover about half of the overall functionality required of the on-board control computer, the other half being devoted to attitude determination and control.

The PUS defines a standard protocol for the transmission and distribution of telecommands (TC) on board the spacecraft and specifies a standard set of services for designers to draw from in the construction of data handling systems.

The PUS defines those standard services by prescribing the service model, i.e. the manner in which application processes on board are to behave on the arrival of any given service request, and the inner structure of the telecommand and telemetry (TM) packets exchanged to this end. The standard nature of the PUS specification considerably facilitates the design for reuse of software components supporting the implementation of compliant systems.

The project that first attempted to exploit the reuse potential of the PUS was denominated OBOSS, for on-board operations support software. OBOSS used Ada 83 [8] as the programming language and inspired to HRT-HOOD for the design method. HRT-HOOD was especially useful to elicit the extent of concurrency that is inherent to the PUS service model. Ada in turn was rich with language constructs that provide the degree of modularity, scalability and genericity that form the crucial basis to any reusable product. As a reflection of the imprint from both sources, the OBOSS concept comprised an orderly collection of active and passive objects along with a domain-specific architecture for their utilisation. In keeping with the capabilities of Ada 83, the OBOSS objects have structure, state and operations but lack true inheritance (in the sense of incremental build) and polymorphism, which are the distinguishing characters of full object orientation. OBOSS objects may be derived from by generic instantiation and are made active by embedding the required type of concurrency (in the HRT-HOOD sense of cyclic, sporadic and protected) within them. These features, if limited in the respect of full object orientation, appear to be sufficient to the achievement of a considerable potential

for unmodified reuse, with important benefits in terms of effort saving.

The emergence of the Ravenscar Profile in 1997 presented the OBOSS project team with the opportunity to revisit certain implementation choices in the area of concurrency and to challenge the expressive power of the profile. OBOSS 2nd generation (OBOSS-II) [10] originated as the team's response to these stimuli. This paper discusses selected aspects of this response. Accordingly, all references to OBOSS in the remainder of this paper specifically denote OBOSS 2nd generation.

An additional dimension of interest is whether and how the Ravenscar Profile integrates with the object orientation capabilities of Ada 95 [9]. The belated appearance of Ada 95 cross compilation systems for space-qualified target processors (cf. e.g.: [5]) has prevented the current OBOSS revision from being able to augment its object oriented characteristics. Yet, the issue is of major importance to the future of OBOSS. Attention is currently being devoted to determining whether and to what extent the exploitation of inheritance and polymorphism of Ada 95 could add to the expressiveness and reusability qualities of OBOSS without detracting from the current integrity and predictability of the system.

Along these lines, this paper addresses three aspects that especially characterise the present and the future of the OBOSS concept: section 2 presents the approach we took to embedding Ravenscar-compliant concurrency in OBOSS; section 3 shows that the expressive power of the profile is indeed sufficient to meet the requirements of systems in the OBOSS application domain; section 4 briefly discusses present and future flavours of object orientation in OBOSS.

## 2. Embedding Ravenscar Concurrency

### 2.1. Motivations

The tasking model that emanates from the Ravenscar profile is particularly restrictive in terms of what it excludes of the wealth of tasking constructs supported by the language definition. Yet the profile is powerful enough to express a plausible implementation of the cyclic, sporadic and protected objects defined by HRT-HOOD, which are among the main ingredients of the OBOSS concept. The prime motivation behind the definition of the Ravenscar Profile was the quest for memory size and execution time efficiency, predictability and verifiability. This notion made an especially compelling argument for OBOSS to consider, for OBOSS addresses an application domain constantly seeking those characteristics. We faced two major challenges in the decision to shape the OBOSS concurrency model in accord with the Ravenscar Profile. The first challenge was to express this concurrency in a manner that would fit in the

overall design philosophy of the reuse library. The second challenge was to determine what drawbacks and limitations, if any, would arise for an OBOSS-based system from the restrictions imposed by the profile. The remainder of section 2 will present the response to the first challenge. Section 3 will discuss the issues arising from the second challenge.

### 2.2. OBOSS Cornerstones

The OBOSS reusable software components belong in three main categories: (i) classes<sup>1</sup>, implemented as generic units, which the user derives from by way of generic instantiation; (ii) templates, implemented as model library units, which the user must tailor (i.e. source edit) to suit the specific needs of the system; (iii) auxiliary objects, which OBOSS uses for the definition of class and template objects, and that the user need not access. OBOSS objects in any of these three categories are either active or protected or passive in the way HRT-HOOD defines these notions.

Passive objects are composite (i.e. parent) objects the implementation of which needs child objects of passive type only. Active objects are composite objects that are implemented by at least one active child object. Child objects may in turn be parent or terminal. Terminal objects have no child. Passive terminal objects do not possess an own thread of control and their methods (called exported operations in HOOD parlance) are executed by the calling thread of control. Active terminal objects possess their own thread of control, which may be either cyclic or sporadic. Cyclic terminal objects provide no methods for other objects to invoke, hence they only invoke the methods of others. Sporadic terminal objects provide one single method (notionally denominated Start) which other objects invoke to trigger the sporadic operation of the object. The Start method is asynchronous in that it returns immediately after delivering the intended execution request to the mailbox of the designated service object. Active objects that provide methods other than Start have them implemented by child objects that are either passive or protected. Protected objects provide methods that warrant synchronous and mutually exclusive execution.

The structure of the active terminal objects in OBOSS rigorously conform to the prescriptions of the Ravenscar Profile. The most natural way for OBOSS to factor out this structural conformance was to represent sporadic, cyclic and protected terminal objects as class objects. In this manner, individual active terminal objects are simply obtained by generic instantiation of the corresponding class object. To illustrate the approach, in the following we present the implementation of the class objects that make up a sporadic terminal object.

---

<sup>1</sup>We use the notion of class in a loose sense: an OBOSS class may instantiate to a class object, but cannot, as yet, be extended.

The requirement that sporadic terminal objects provide one single method denominated `Start` fully determines the specification of the corresponding generic unit:

```
with ... – auxiliary definitions including:
with Priority_Ctrl; – system-specific priority definition
generic
... – configuration parameters including:
Task_Priority : in Priority_Ctrl.Active_Task_Priority;
Task_Stack_Size : in Natural;
Event_Buffer_Priority : in Priority_Ctrl.Protected_Priority;
Event_Buffer_Size : in Positive := < ... >;
type Param_Type is private;
with procedure Operation (Param : in Param_Type);
– sporadic operation
package Sporadic_Task is
function Start (Param : Param_Type) return Boolean;
end Sporadic_Task;
```

OBOSS active components communicate by message passing, as a reflection of which the parameter of the `Start` operation is expected to be a packet-encoded message with a variable structure. The type of the parameter determines the type of the incoming message, that is to say the inner structure of the corresponding packet. The sporadic operation of the task (`Operation`) takes a parameter of the same type as the `Start` operation, which allows the former to operate on the incoming message. The arrival of the message defines the triggering event for the sporadic task. The `Start` operation in OBOSS and in HRT-HOOD is asynchronous (i.e. non-blocking) and, thus, it simply posts the message in a dedicated event buffer. OBOSS factors out the definition of this buffer too, and implements it as an auxiliary class protected object. Every instantiation of a sporadic task will have its own event buffer with a given size (`Event_Buffer_Size`) and a given ceiling priority (`Event_Buffer_Priority`).

```
with ... – useful definitions including:
with Queue; – auxiliary class object for event buffer
with System_Clock; – interface to system clock
package body Sporadic_Task is
package Event_Buffer is new Queue
(Element_Type => Param_Type,
Queue_Size => Event_Buffer_Size,
Queue_Priority => Event_Buffer_Priority);
function Start (Param : Param_Type) return Boolean is
begin
return Event_Buffer.Deposit (Param);
end Start;
task Sporadic_Task is
pragma Priority (Task_Priority);
pragma Storage_Size (Task_Stack_Size);
end Sporadic_Task;
task body Sporadic_Task is
Due_Event : Param_Type;
Activation_Time : Ada.Real_Time.Time := ...;
```

```
begin
delay until Activation_Time;
loop
Due_Event := Event_Buffer.Extract;
Operation (Due_Event);
end loop;
exception
when ... – appropriate handling
end Sporadic_Task;
end Sporadic_Task;
```

All the sporadic task is to do at this level of specification is to fetch one message off the event buffer and perform the appropriate operation on it. The OBOSS implementation allows for the occurrence of exceptions and performs some standard handling on them. (Cf. section 3 for a discussion of this issue.) Partly owing to limitations with the language support in use and partly to design decisions, the current OBOSS implementation ties the exception handling policy at this level to the specific OBOSS build.

The implementation of the class object `Queue`, from which the protected terminal object `Event_Buffer` is derived, determines the type of data-oriented synchronisation model in place between the caller of the `Start` operation and the sporadic task itself. In fact, the current implementation renders the `Start` operation non-blocking even on buffer full, while it blocks the sporadic task indefinitely on the `Extract` operation on buffer empty. (Cf. section 4.1 for a discussion of data-oriented synchronisation issues in OBOSS.)

At present, both the implementation of the queue data structure and that of the `Deposit` and `Extract` operations are fixed for any specific OBOSS build. Typical choices for the insertion and extraction policy are FIFO or priority based.

```
with ... – auxiliary definitions
generic
type Element_Type is private;
Queue_Size : in Natural;
Queue_Priority : in Priority_Ctrl.Protected_Priority;
package Queue is
function Deposit (Elem : in Element_Type)
return Boolean;
function Extract return Element_Type;
end Queue;
package body Queue is
protected Queue is
pragma Priority (Queue_Priority);
procedure Deposit (Elem : in Element_Type;
Response : out Boolean);
entry Extract (Elem : out Element_Type);
private
– implementation details
end Queue;
function Deposit (Elem : in Element_Type)
return Boolean is
Response : Boolean;
begin
```

```

Queue.Deposit (Elem, Response);
return Response;
end Deposit;
function Extract return Element_Type is
Elem : Element_Type;
begin – Extract
Queue.Extract (Elem);
return Elem;
end Extract;
– implementation details
end Queue;

```

### 2.3. Building an Application Process

A PUS system may be regarded as a collection of independent and occasionally cooperating application processes. Application processes represent physical or logical on board subsystems, which may be commanded into the provision of a number of specific services.

An OBOSS application process is easily built up by composition and configuration of reuse modules according to a recurrent template, which we illustrate in the following. At the top of the OBOSS hierarchy we encounter the notion of `Application_Process`, the body of which defines the exported operation `Send_TC` in terms of an operation with the same profile exported by a service unit (a child object in HRT-HOOD parlance):

```

with PUS; – OBOSS reuse component
with ... – non-OBOSS components
package Application_Process_j is
... – non-PUS components
procedure Send_TC (Packet : in PUS.Packet);
– for message producers to invoke
end Application_Process_j;

with AP_j_Dispatcher – to handle incoming messages
with ... – supplementary components
package body Application_Process_j is
... – for non-PUS components
procedure Send_TC (Packet : in PUS.Packet) is
begin
AP_j_Dispatcher.Send_TC (Packet);
end Send_TC;
end Application_Process_j;

```

As we resolve the message dispatching component of `Application_Process_j` (i.e. `AP_j_Dispatcher`) we encounter the first instance of embedded concurrency within the OBOSS system. The specification of the dispatching object is rather obvious and very symmetrical to the specification of its parent object. The body of this package in particular shows how OBOSS makes certain objects active by attaching the desired concurrent behaviour to their definition:

```

with PUS; – OBOSS reuse component

```

```

package AP_j_Dispatcher is
procedure Send_TC (Packet : in PUS.Packet);
end AP_j_Dispatcher;

with ... – OBOSS reuse components including:
with AP_j_TC_Interpreter;
with AP_j_Params; – configuration parameters
package body AP_j_Dispatcher is
package TC_Forwarder is new Sporadic_Task
(Task_Priority => AP_j_Params.Task_Priority,
Task_Stack_Size => AP_j_Params.Task_Stack_Size,
Event_Buffer_Priority => AP_j_Params.Buffer_Priority,
Event_Buffer_Size => AP_j_Params.Buffer_Size,
Operation => AP_j_TC_Interpreter.Receive_Packet,
... – all other parameters);
procedure Send_TC (Packet : in PUS.Packet) is
begin – non-blocking
if not TC_Forwarder.Start (Packet)
... – error handling on buffer full
end if;
end Send_TC;
end AP_j_Dispatcher;

```

We note from the above code fragments that a call to `Application_Process_j.Send_TC` resolves into a call to the `Start` operation of the sporadic task `TC_Forwarder` embedded in `AP_j_Dispatcher`. We have seen earlier that the implementation of the `Start` operation entails a transaction on the local event buffer embedded within the sporadic task instance. The implication of this choice is that the message sender is fully decoupled from the message receiver, so that they may effectively proceed in parallel. The key notion we derive from the implementation of procedure `Send_TC` is that OBOSS lets the sender proceed even in case of buffer full. The implementation traps the failure event without blocking the sender and produces the appropriate response, which typically amounts to a failure report destined to the sender and/or to some supervisory authority.

The style that OBOSS has chosen to embed concurrency within objects allows the user to capture in the form of generic actual parameters such notions as: the operation that the embedded thread is to carry out; the priority at which that is to be done; the ceiling priority and the size of the associated event buffer (if any) or any other relevant service data structure. This shows for example in the instantiation of sporadic task `TC_Forwarder` that we have seen in the code fragment above.

The concurrent operation of `TC_Forwarder` is defined by package `AP_j_TC_Interpreter`. The specification of this definition package simply exports the procedure nominated as generic actual parameter of `TC_Forwarder`. Procedure `Receive_Packet` effectively multiplexes messages (i.e. service requests) to the appropriate service provider within the application process. This is in keeping with the delegation of control approach taken by OBOSS, which allows service

providers to be implemented as active objects and thus to operate concurrently.

```
with PUS; – OBOSS reuse component
package AP_i_TC_Interpreter is
  procedure Receive_Packet (Packet : in PUS.Packet);
end AP_i_TC_Interpreter;
```

```
with ... – OBOSS reuse components including e.g.:
with AP_i_Service_J;
  – instantiation of PUS service J in application process i
package body AP_i_TC_Interpreter is
  procedure Reject_Packet (Packet : in PUS.Packet) is
    ... – to handle erroneous messages
  end Reject_Packet;
  procedure Receive_Packet (Packet : in PUS.Packet) is
  begin
    case Get_Service_Type (Packet) is
      – for any supported PUS service
      when Service_J =>
        AP_i_Service_J.Receive_Packet (Packet);
      – for all unsupported services
      when others => Reject_Packet (Packet);
    end case;
  end Receive_Packet;
end AP_i_TC_Interpreter;
```

One level below application processes in the OBOSS hierarchy we encounter the implementation of the PUS service capabilities to embed within application processes. This is where OBOSS pushes its reuse approach to its extreme.

AP\_i\_Service\_J is described as an active (sporadic) object, which in HRT-HOOD parlance means that it may embed multiple threads with predominant (i.e. externally visible) sporadic behaviour.

The implementation of AP\_i\_Service\_J.Receive\_Packet involves the concurrent operation of (at least) one sporadic task. This implementation approach warrants loose coupling between the message reception mechanism at application level and the execution of the service protocols supported by the application process.

The generic unit Service\_J fully defines the structures and operations corresponding to PUS service J and therefore suffices for the implementation of the service instantiation embedded within the application process.

```
with Service_J; – OBOSS reuse component for service J
with AP_i_Router; – to send response messages
with ... – any non-OBOSS components
package AP_i_Service_J is new Service_J
  (Deposit => AP_i_Router.Deposit; ...);
end AP_i_Service_J;
```

```
with ... – OBOSS reuse components
generic
  ... – service-specific configuration parameters
```

```
with Deposit (Packet : in PUS.Packet) return Boolean;
package Service_J is
  procedure Receive_Packet (Packet : in PUS.Packet);
  – for service request to reach service provider
end Service_J;
```

OBOSS uses the instantiation of the formal generic operation Deposit to dispatch the messages produced by the operation of the service object to the designated destination mailbox. Because of its inherent inner concurrency, Service\_J is an active parent object in the HRT-HOOD sense. Its implementation encompasses a considerable amount of recurrent active and passive components:

```
with ... – OBOSS reuse components
package body Service_J is
  ... – auxiliary instantiations
  package Cmd_Interpreter is
    – command reception concurrent to service execution
    procedure Receive_Packet (Packet : in PUS.Packet);
  end Cmd_Interpreter;
  procedure Receive_Packet (Packet : in PUS.Packet) is
  begin
    Cmd_Interpreter.Receive_Packet (Packet);
  end Receive_Packet;
  package body Cmd_Interpreter is
    ... – auxiliary definitions
    procedure Execute_Cmd (Packet : in PUS.Packet);
    package Interpreter is new Sporadic_Task
      (Operation => Execute_Cmd, ...);
    procedure Receive_Packet (Packet : in PUS.Packet) is
    begin – caller does not block
      if not Interpreter.Start (Packet)
        ... – error handling on buffer full
      end if;
    end Receive_Packet;
  end Cmd_Interpreter;
end Service_J;
```

The very same approach taken to the embedding of concurrency in the handling of messages across and within application processes applies to the execution of the specific service protocols. We note in fact that the implementation of the exported operation Receive\_Packet of unit Service\_J resolves in depositing the service request into the event buffer of Cmd\_Interpreter.Interpreter. In this way the execution of the service protocol within any application process may proceed concurrently to the dispatching of messages within the system.

It is also a prerogative of the OBOSS design that the processing of the service request carried out by the concurrent operation Execute\_Cmd of sporadic task Cmd\_Interpreter may produce messages that trigger the operation of multiple concurrent threads both within the service instance itself and without it. This concurrency potential is obtained through the same mechanism that warrants concurrency of

execution between the Send\_TC operation invoked by a message producer outside Application\_Process\_I and the Receive\_Packet operation that dispatches the message to its internal destination.

Individual application processes in OBOSS may embed multiple message producers. The architecture of OBOSS requires that all outgoing messages get into the mailbox of a centralised packet router, the sole entity with knowledge of all legal destinations in the system. To this end, a dedicated passive object per application process collects all outgoing messages and dispatches them to the mailbox of the packet router. The base class for this dispatching object is denominated Packet\_Depositor. The dispatching operation is simply implemented by: (1) connecting the Deposit operation exported by the corresponding class object to the mailbox of the packet router; and (2) using the resulting operation for the instantiation of the formal generic Deposit operation of the message producer concerned (e.g.: AP\_i\_Service\_J).

OBOSS leverages on the recurrence of the activity performed by the Packet\_Depositor to yield another obvious instance of reusable software component.

```

with Packet_Depositor; -- OBOSS reuse component
with Packet_Router; -- the system-level message router
package AP_i_Router is new Packet_Depositor
  (Deposit_Response => Packet_Router.Deposit);

with PUS;
generic
  with function Deposit_Response (Packet : PUS.Packet)
    return Boolean;
package Packet_Depositor is
  function Deposit (Packet : PUS.Packet) return Boolean;
end Packet_Depositor;

with ... -- OBOSS reuse components
package body Packet_Depositor is
  ... -- service operations including:
  function Deposit (Packet : PUS.Packet)
    return Boolean is
  begin .. -- administration on outgoing Packet
    return Deposit_Response (Packet);
  end Deposit;
end Packet_Depositor;

```

### 3. The Ravenscar Profile as a Limiting Factor

#### 3.1. Understanding the Profile Restrictions

The Ravenscar Profile defines a subset of the Ada 95 language that aims at predictable and efficient use of concurrency in high-integrity real-time systems. Concrete implementations of the profile, however, tend to introduce additional restrictions with a view to further reducing the causes

of potential non-determinism at run time as well as to comply with the prescriptions of specific safety-critical standards (cf. [12]). In particular, specific implementations may choose to prohibit the use of declarations that involve allocations or deallocations from the heap. This is for example the case with explicit use of allocators for access types as well as implicit allocations generated by the compiler, for example as a result of array slicing.

Exception handling, about which the profile definition currently is silent, is another issue on which concrete implementations may adopt restrictive strategies. The problem with exception handling in the general case is that the time to transfer control to the appropriate handler on the raising of an exception at run time may be difficult to bound. This event may present an hazard to the system and thus contrast with the spirit of the profile. Restrictive implementations may prohibit the use of user-defined exceptions altogether and only support the occurrence of predefined exceptions. The handling of predefined exceptions would then simply involve direct transfer of control to designated 'last-rites' routines explicitly supplied by the user.

While specific design and analysis efforts should be directed to preventing the occurrence of predefined exceptions, all systems that take input from inherently unsafe sources are intrinsically exposed to exception situations at run time. On-board systems like OBOSS definitely belong in the latter category.

Two questions are crucial to determine the nature of the exception handling support suited for an application with integrity requirements: (i) whether static analysis can prove that the time taken at run time to locate and reach the designated handler does not pose hazards to the system; and (ii) whether an appropriate level of treatment is possible for any given exception.

Question (ii) above only allows an application-specific response, whereas question (i) takes a positive, application-independent answer. Research work (cf. e.g.: [4]) has in fact shown that a few minor restrictions suffice to render worst-case timing analysis of the Ada exception handling mechanism tractable. Arguably, the reduced tasking model that emanates from the Ravenscar Profile further simplifies the exception propagation mechanism, thus lending itself to useful and analysable exception handling models. The very design of OBOSS specifically reflects this view.

#### 3.2. Implications on OBOSS

Overall, the prescriptions of the Ravenscar Profile did not present OBOSS with significant implementation problems. In fact, the coding restrictions that emanate from the profile are not distant from the standard coding guidelines normally followed by a large fraction of the on-board software community and, ironically, even more permissive with

respect to the use of native tasking.

The prescription for task declarations to place at library level only, was comparatively natural for OBOSS to comply with. In fact, all of the package nesting that so distinctly marks the OBOSS design outlined in section 2 rigorously occurs at library level. Within this structure, the only OBOSS packages that define task objects do so at library level too, as shown for package `Sporadic_Task` and `sit` at the bottom of the nesting hierarchy, whereby any such tasks transitively become library-level objects in their respective chain of instantiation.

Heap operations at run-time do indeed pose a problem to systems in the application domain of OBOSS. The non-deterministic worst-case execution times resulting from a fragmented heap is a serious threat to the predictability of the response time behaviour of the system. In the holy name of abstraction, OBOSS occasionally uses declarations and constructs that the compiler may resolve by insertion of implicit heap operations. In particular, OBOSS uses discriminated records and unconstrained array types to provide encompassing abstractions for data structures that are polymorphic by nature. This is for example the case with the PUS packets handled by the OBOSS objects, which have variable size and variable inner structure by their very definition. Most compilers would allocate such polymorphic objects on the heap, and even create multiple copies of them, for example when they are returned as the result of subprogram calls. On consideration of the potential hazard of non-deterministic execution against the expressive power (and beauty) of the abstraction, the OBOSS team is now considering to eliminate any constructs involving dynamic heap management from future releases of OBOSS.

Prohibiting the use of exceptions altogether would be too restrictive and thus unacceptable for OBOSS. Exceptions offer an excellent means for separating the management of error situations from that of nominal situations, and, as such, they are extensively used throughout OBOSS. In fact, the OBOSS design philosophy accepts that certain operations may fail at run time and requires that those should report the failure event in the form of a specific exception. Notable examples of such operations in OBOSS are those that have to take data input from unreliable sources and those that have to cope with user demands in the event of temporary shortage of resources. In order to ensure that the control flow modification resulting from exception events be always statically analysable and would not induce hazards to the system, OBOSS requires that operations that may raise exceptions be encapsulated within blocks equipped with appropriate handlers. In fact these prescriptions are in keeping with the provisions that [4] assumes for the static timing analysis of Ada exception handlers.

Abandoning exceptions would imply that the vast majority of subprograms in OBOSS would have to return an enu-

meration type indicating any possible error status resulting from the call. This would detract from the clarity of the control flow by clogging it with conditional checks that propagate the result value up the call graph to the level where the error were to be handled.

## 4. OBOSS Flavours of Object Orientation

### 4.1. Data-oriented Synchronisation Issues

Concurrent objects typically cooperate by exchange of messages and so do OBOSS objects too. The processing of the message flow produced by concurrent objects may give rise to undesirable interactions. Implementations use various forms of object synchronisation to prevent the occurrence of such events.

OBOSS uses synchronisation as the designated means for objects to exchange data. More specifically, OBOSS treats the arrival of an incoming message as the activation event of the destination object. For OBOSS thus we talk of data-oriented synchronisation, which is one particular manifestation of the broader concern of the synchronisation of concurrent objects.

The synchronisation of concurrent objects effectively amounts to verifying specific plausibility conditions for the processing of individual messages in a message flow. [7] distinguishes three broad categories of synchronisation constraints that determine such conditions: (i) exclusion constraints, which hold when implementation characteristics of the objects concerned inhibit concurrent processing of multiple messages; (ii) state constraints, which hold when the internal state of an object prevents the correct processing of messages; (iii) coordination constraints, which involve the coordination of multiple messages to multiple destinations.

We talk of server-side synchronisation when the applicable constraints are enforced as part of the message acceptance mechanism of the destination object. Conversely we talk of client-side synchronisation as the constraint is enforced on the sending of the message.

Arguably, the adoption of server-side synchronisation directly follows from the use of the Ravenscar Profile in force of which protected objects operate as servers to communicating tasks and protected operations naturally support the enforcement of the applicable synchronisation constraints.

An obvious consequence of demanding the enforcement of synchronisation constraints is that the object behaviour in the event of their violation must also be specified. [7] defines three classes of response behaviour on violation of synchronisation constraints: (i) balking, which is to return immediately with a failure indication; (ii) blocking, which is to wait indefinitely until the constraint is met; (iii) timed waiting, which is to wait for up to a maximum time

for the constraint to be met, returning with an error otherwise. Whereas the Ada language definition [9] supports all three classes of response behaviour, the Ravenscar Profile excludes the use of timed-wait constructs and limits the use of blocking to at most one waiter per protected entry queue.

OBOSS contemplates exclusion constraints and state constraints while it currently has no requirements for the support of coordination constraints. The use of protected objects for the implementation of event buffers allows OBOSS to enforce exclusion constraints in the simplest and most radical form of total mutual exclusion. State constraints hold for OBOSS synchronisation in the classical events of buffer full or buffer empty.

The use of HRT-HOOD led OBOSS to model message receivers as sporadic objects and the message arrival as the activation event. As the design method requires, the exported operation `Extract` of base class `Queue` shown in section 2.2 maps to a protected entry with blocking response on state constraint violation of buffer empty. For state constraint violation of buffer full on sending, instead, OBOSS has opted for a balking response, in keeping with the Ravenscar Profile. Accordingly, the `Deposit` method of `Queue` maps to a protected procedure that returns a boolean response parameter. A negative response on sending denotes a state constraint violation of buffer full and results in the issue of an exception message directed toward a supervisory authority. This behaviour is in accord with what is typically expected of current on-board systems. In section 2.3 we have encountered two code fragments that give rise to balking response behaviour: in the implementation of operation `Send_TC` of object `AP_i_Dispatcher`; and in the implementation of operation `Receive_Packet` of object `Cmd_Interpreter` within class object `Service_J`.

It is our opinion that the synchronisation behaviour of OBOSS objects fully meets the relevant requirements of the target application domain. This observation arguably proves that the expressive power of the Ravenscar Profile in this particular respect is sufficient for space applications in the domain of OBOSS.

## 4.2. Other OO Issues

OBOSS class objects can be instantiated but cannot presently be extended. This limitation bounds the reusability potential of OBOSS and weakens the object orientation characteristics of its design.

The class object `Queue` that we introduced in section 2.2, presents an obvious case where inheritance by extension would be more powerful and advantageous than mere inheritance by instantiation. The current design of the object, which is central to the data-oriented synchronisation characteristics of OBOSS, in fact suffers from two intrinsic limitations (cf. [2]):

1. The implementation of the object is fixed for any given definition of the internal data structure and of the exported operations `Deposit` and `Extract`: a re-definition of the whole module (and thus another OBOSS build altogether) would therefore be required to support any other implementation.
2. Any given instantiation of the object fully determines the data type that the data structure holds and that the exported operations are able to manipulate (i.e. `Element_Type`): no operation may be thus defined that manipulates the data structure without regard to the actual data stored in it.

Potential OBOSS users may well wish to specify for any group of sporadic objects in the system the desired insertion and extraction policy for use with the respective event buffer. With the current solution instead the choice (e.g.: FIFO, priority based) is irremediably fixed for all objects for any OBOSS build. This is no intolerable drawback at present, yet it may become more of an annoyance in the face of increasingly demanding users. It is a subject of current investigation the issue whether OBOSS include other components as obviously and as conveniently amenable to extension as the event buffer. A further issue of concern is the increase in resource requirements (particularly code size) that may arise from the adoption of inheritance by extension.

Another opportunity for OBOSS to benefit from the increased expressive power of Ada 95 may obviously arise from the use of tagged types in the place of discriminated records for the handling of polymorphic data objects. The case for replacement is particularly attractive from the abstraction point of view. It remains to be seen however whether compilers would treat objects of tagged type in a less heap-intensive way than they tend to do for discriminated records in the use context of OBOSS. Any increase in the heap-intensiveness of their treatment would obviously detract from their ultimate appeal.

Under all circumstances and in spite of the intellectual interest of full object orientation, OBOSS revisions will always give precedence to the preservation of the current integrity, predictability and resource demand of the system.

## 5. Overall Assessment and Future Directions

Fitting the Ravenscar Profile in the original OBOSS baseline proved fairly smooth and, in retrospect, particularly natural and convenient to the overall economy of the OBOSS concept. This notion arguably shows that the profile suits the needs of on-board real-time embedded systems of the new generation very well.

The combined effect from the OBOSS reuse philosophy and the Ravenscar tasking model calls for a radical 'special-

isation' of tasks, for tasks can only exhibit a very specific execution behaviour and thus can only adequately perform specific operations. This notion adds to the clarity of design but also increases the tasking population of the system to unprecedented levels, which demands an unusually large range of priorities.

One inescapable consequence of this phenomenon is that Ravenscar-compliant compilers may have to support much larger priority ranges than usually contemplated. (By way of example, an average OBOSS system would comprise no less than 40 tasks and 40 protected objects, thus needing up to 80 distinct priorities.) The use of distinct priorities is not a necessary requirement to the construction of a predictable real-time system. Arguably, however, it adds to the simplicity of the runtime implementation as well as to that of the scheduling analysis.

The prescription to avoid the dynamic use of heap brought to the foreground the need to thoroughly weigh the beauty of abstraction against any ensuing breach to integrity and predictability. This notion proved a useful lesson to the OBOSS team. Future releases of the product will eliminate all constructs that violate the prescription, for this will further increase the ultimate value of the system.

The OBOSS experience suggests that evolutions or interpretations of the Ravenscar Profile should neither contemplate nor encourage the prohibition of user-defined exceptions. In section 3.2 we argued against the increase in complexity and obfuscation of program logic that may result from the lack of user-defined exceptions. Exceptions are a delicate matter for all systems with integrity requirements, and their presence or absence and their handling are critical issues for design, implementation and verification. OBOSS found what we consider an acceptable way of coping with the unavoidable presence of exceptions in our application domain. Other designers and other application domains may perhaps find the OBOSS approach not appropriate to their needs. Without detracting from this diversity of positions, we argue that the Ravenscar Profile should possibly be annexed the definition of model(s) for the use of exceptions that preserve the spirit and the intent of the profile.

The interaction between the Ada concurrency (and the Ravenscar Profile in particular) and object orientation constitutes another dimension of considerable intellectual interest. The prime goal of OBOSS is to maximise its reuse potential. True object orientation can thus be of great benefit to it. The current release of OBOSS uses Ada 83 class objects in the form of generic units. Such objects can be instantiated but cannot be extended. In this paper we have briefly discussed one typical case in which object extension would be of obvious benefit to OBOSS.

The present highlight of OBOSS lays in the merit of its virtually full compliance to the Ravenscar Profile. The fu-

ture direction of this product will most certainly lay in the acquisition of a more mature object oriented connotation, in a fashion that preserve the integrity, predictability and resource demand of the system.

**Disclaimer.** The views expressed in this paper are those of the authors' only and do not necessarily engage those of the European Space Agency.

## References

- [1] A. Burns. The Ravenscar Profile. *Ada Letters*, XIX(4):49–52, 1999.
- [2] A. Burns and A. Wellings. *Concurrency in Ada*. Cambridge University Press, 1988. Second edition.
- [3] A. Burns and A. Wellings. *HRT-HOOD: A Structured Design Method for Hard Real-Time Systems*. Elsevier Science, Amsterdam, NL, 1995. ISBN 0-444-82164-3.
- [4] R. Chapman, A. Burns, and A. Wellings. Worst-case timing analysis of exception handling in Ada. In L. Collingbourne, editor, *Proceedings of the Ada UK Conference*, pages 148–164. IOS Press, The Netherlands, 1993.
- [5] ESTEC. 32-bit Microprocessor and Computer System Development. Study Contract No. 9848/92/NL/FM, European Space Agency, Noordwijk, NL, 1996-1999. (<http://www.estec.esa.nl/wsmwww/erc32/erc32.html>)
- [6] European Space Agency, Noordwijk, NL. *Packet Utilisation Standard*, May 1994. PSS-07-101 Issue 1.
- [7] D. Holmes, J. Noble, and J. Potter. Effective Synchronisation of Concurrent Objects: Laying the Inheritance Anomaly at Rest. Technical report, Department of Computing, Macquarie University, Sydney, AU, 1998. (<http://www.mri.mq.edu.au/~dholmes/research/oopsla-paper-submission.html>).
- [8] ISO. *Ada Reference Manual*. International Standardisation Organisation, Geneva, CH, 1987. ISO/IEC 8652:1987.
- [9] ISO. *Ada Reference Manual*. International Standardisation Organisation, Geneva, CH, 1995. ISO/IEC 8652:1995.
- [10] TERMA Elektronik. Software System Development for Spacecraft Data Handling Control. November 1999. Product of ESTEC Contract 12797/98/NL/PA. ([http://spd-web.terma.com/Projects/OBOSS/Home\\_Page](http://spd-web.terma.com/Projects/OBOSS/Home_Page)).
- [11] T. Vardanega and J. van Katwijk. A Software Process for the Construction of Predictable On-Board Embedded Real-Time Systems. *Software - Practice and Experience*, 29(3):235–266, 1999. John Wiley & Sons.
- [12] Wichmann, B. Programming Languages - Guide for the Use of the Ada Programming Language in High Integrity Systems. Technical Report ISO/IEC TR 15942 (originated as ISO/IEC JTC1/SC22 WG9 N359), International Standardisation Organisation, Geneva, CH, March 2000. (<http://www.dkuug.dk/jtc1/sc22/wg9/documents.htm>).