Ravenscar Design Patterns? Reflections on Use of the Ravenscar Profile

Tullio Vardanega
University of Padua
Dept. of Pure and Applied Mathematics
via G. Belzoni 7, 35131 Padua, Italy
tullio.vardanega@math.unipd.it

Abstract

Industrial developers have started to use the Ravenscar Profile in challenging application domains. The thorough feedback that begins to emerge from these experiences is of great interest to the community (i.e. the IRTAW group) that first conceived and promoted the Profile. This paper undertakes a first analysis of a specific experience report to which the author had early and privileged access.

1. Introduction

A paper recently appeared in the proceedings of the Reliable Software Technology — Ada Europe 2003 conference [5] reports on an industrial experience on the use of the Ravenscar Profile in the development of the control software embedded on board a new Earth observation satellite, named GOCE, soon to be launched by the European Space Agency. Such a report is especially attractive in that it provides very thorough feedback from the use of the Profile in a challenging field application.

This paper aims to reflect on what the author regards as the most critical aspects of the reported user experience. The author's reflections were considerably facilitated by the close contact he enjoyed with the GOCE platform application software (PASW) development team.

The intent of this paper is to extend the feedback cycle to the IRTAW forum where the Ravenscar Profile was designed, specified and promoted, with a view to possibly enhancing the guidance material [1] that currently accompanies the specification of the Profile.

2. Problem Outline

Two fundamental factors appear to have determined the GOCE PASW architecture: (i) the hardware configuration of the platform to be controlled; and (ii) the designer's choice to follow the Ravenscar Profile restrictions in a form augmented with two further constraints that we will discuss later in this paper, so that the resulting software system should be more easily amenable to static scheduling analysis.

Factor (i) is typical of any embedded system. Yet, as a reflection of its specific domain of application, the GOCE system architecture presents distinguishing challenges that stem from its predominantly data-flow oriented nature. In particular, the developer views the following aspects of the GOCE system architecture as crucial:

R1. the system includes a range of physical I/O channels that multiplex data and messages of several types to and from a variety of devices with differing real-time requirements and which require the use of a *suspensive event*, either an interrupt or a time delay, to synchronise the acquisition of the read-out

R2. the I/O channels are *shared* among concurrent tasks, which may therefore compete for the execution of multiple data transfers over the same channel

R3. both the GOCE PASW and its peripheral units have limited and costly *buffering* resources and the system allows no data to be lost (lest the mission product decrease), which implies that the size of data buffers must kept as small as possible and buffer overflow must be avoided.

Factor (ii) was the free choice of the designer in anticipation of an incoming domain-specific process requirement that attributes competitive advantages to software systems that be statically verifiable as well as be designed for efficiency and reliability.

As stated in [1], the motivation for the Ravenscar Profile was specifically to enable the construction of efficient, reliable and verifiable concurrent real-time programs, in perfect match with the designer's need above.

The Profile restrictions trade dramatic reductions in the complexity of the compiler and the associated run-time sys-

tem for reliability of implementation and efficiency of execution. The same restrictions, however, were also designed to facilitate off-line verification (notably scheduling analysis) of the concurrent behaviour of the program. Various techniques currently exist for scheduling analysis, which range from fairly straightforward to very complex.

In fact, there exists an inverse relation between the complexity of the analysis technique and the strictness of the coding style: the simpler the desired analysis the stricter the required style; which was the motivation for the further restrictions adopted in the GOCE design.

According to the developer, the combined effect of the two factors above outlined had a profound impact on the GOCE software architecture. This paper performs an initial analysis of the reasons for this impact and the implications that may be drawn from it.

Ultimately, the developer's feedback revolves around two key observations, which both arise from the way factor (ii), the chosen design paradigm, interacted with factor (i), the hardware dependence:

- an unexpected impact on the algorithm design compounded the paradigm shift imposed on the design of the system architecture; impact from the latter was anticipated in the transition from the earlier liberal and unconstrained design approach (conducive to possibly inordinate verification cost) to the severely-constrained one resulting from adoption of the Ravenscar Profile; impact from the former was instead unexpected and caused some design hurdles

- a laborious deployment of the static timing analysis process from its inception, with the capture and apportionment of the real-time requirements and attributes onto the application and its design components, down to verification, with specific tools for worst-case execution time analysis [8, 3] and scheduling analysis (outlined in [9]). Both tools were developed under contract with the European Space Agency as part of their technology research and development programme; the net effect of this difficulty was increased cost of — and thus lower return from — the innovative and virtuous design and verification process undertaken by the developer.

The core of the problem seems to lie in a potential conflict between requirements R1-3 and the computational model entailed by the scheduling analysis tools selected by the designer.

The computational model in question, in addition to fully adhering to the Ravenscar Profile, brings forward two further strict requirements:

R4. every task must have a *single suspension point*; and

R5. every protected entry (which cannot be more than one per protected object) must have a *single task caller*

The former restriction enforces sharp design-level separation between periodic (with absolute time as triggering event) and sporadic (with an other-than-time signal as triggering event) tasks. The latter constructively ensures that no more than one task can ever suspend on the single entry of any protected object.

These two additional requirements were pinpointed as the cause of the most design difficulties. These requirements are not part of the Ravenscar Profile per se. Instead, they can be viewed as direct corollaries of it in any supporting design methodology (e.g.: HRT-HOOD [2] and derivatives [6, 7]) which aims to warrant verifiability by construction, without requiring the use of exceedingly complex analysis techniques. Hence, the two additional restrictions arguably belong in the Ravenscar application domain.

3. Problem Analysis

3.1. Early Capture of Suspension Points

Problem Outline. An important effect on the GOCE PASW design arises from matching the data-flow intensive architecture of the system with the Ravenscar-specific requirement to capture, early in the design process, all potential task suspension points. Each such suspension point carries an activation event for the task concerned.

The developer argues that the imposed match between suspensive interfaces and the single activation event allowed per task causes the software design to strongly depend on the hardware architecture, and specifically on whether the peripheral units controlled by the on-board software are memory mapped or else connected to I/O channels (as is the case with GOCE).

According to the developer, any changes in the way hardware units may be interacted with can thus no longer be concealed in low-level procedures, but it is bound to impact on the concurrent architecture of the system in so far as the changes affect either the nature or the distribution of task suspension points.

Commentary. In the author's opinion, this observed effect, far from being negative, is the inevitable result of a design methodology compliant with the Ravenscar Profile. Any methodologically-supported use of the Profile in fact is bound to *enforce* a design discipline that prevent the user from pushing aside until later (whether deliberately or inadvertently) consideration of physical and/or logical architecture aspects that may have a bearing on the concurrent and timing behaviour of the system. In this view, the capture of any interface that involves a potential suspension point for

the task(s) interacting with it becomes a mandatory activity, prerequisite to the logical design, which tantamounts to the capture of implicit yet crucial requirements on the software architecture.

What may well be necessary to help users fully appreciate the necessity of this discipline is perhaps further and more explicit guidance material on this specific aspect, in complement to what already provided by the current version of the Ravenscar Guide [1].

An interesting research topic that goes beyond the scope of this paper is whether and to what extent the design paradigm that may ensue from adoption of the Ravenscar Profile (with or without further constraints) can be reconciled with the current trend on "aspect-oriented programming", which specifically advocates the very separation of concerns that such a methodological use of the Profile seemingly opposes by imposing a single design viewpoint that demands the early capture of all potential task suspension points.

3.2. Enforcing Single Suspension Points

Problem Outline. A further notable effect on the GOCE PASW architecture stems from the interaction between requirements R1-3 and R4.

The effect in question problem is typified by the PASW interaction with the on-board real-time clock (OBRTC for short). The system uses the OBRTC, which is distinct from the processor clock and not accessible with the normal Ada.Real_Time.Clock function, to time-stamp outbound source data packets.

Not uniquely to GOCE, the OBRTC hardware is located on a peripheral unit connected to the main processor via an I/O channel. Hence, any task `A` wishing to acquire OBRTC read-outs would have to undertake a 3-stage transaction as follows:

1. `A` writes a command to a location in a buffer memory shared between the processor and a (hardware) channel controller, and then suspends itself until notification of the read-out availability;

2. upon reading the command, the channel controller requests the relevant remote device, among possibly several ones, to produce the required data;

3. the channel controller waits for the requested data, which it then stores in a designated buffer memory location and, finally, raises an "I/O complete" interrupt to notify `A` about the data availability.

The logical design that immediately comes to mind to model this transaction would have `A` wait for its "I/O complete' event on a closed entry of a protected channel buffer, which should be opened by the interrupt protected procedure attached to the relevant physical interrupt from the channel.

This model however does not fit GOCE for two related reasons: (i) the link controller acts as the multiplexer of commands and read-outs that connect multiple sources to multiple destinations; (2) the channel-buffer protected object cannot possibly be the one on which *all* client tasks should concurrently wait for their multiplexed event notification, because this would violate the single-waiter Ravenscar restriction.

An alternative design is therefore required, which should separate the user from the provider of OBRTC read-out values. The interaction between the two could be either *reactive*, on demand from the user, or *proactive*, by periodic acquisition with suitable fixed rate, public posting and consequent bounded maximum staleness of clock data.

The GOCE designer adopted the proactive model and noted that requirement R4 challenged the design of the acquisition task, whose intuitive specification features two distinct suspension points: one for the availability signal of the required read-out; the other for the time of the periodic acquisition. The resulting double nature (time-triggered and event-triggered) of the acquisition task, however, is strictly forbidden by the computational model used for GOCE. Notably, the same problem would have arisen for the reactive model too, the request arrival being the second suspension point for the acquisition task.

This challenge is very common and it allows at least two alternative approaches.

One approach is to take one of the two behaviours as dominant. This strategy is described in section 5.7 of the Ravenscar Guide [1] where example 9 outlines an event-triggered task with an inserted time delay used to enforce minimum separation between successive sporadic activations. The approach holds as long as there is a dominant behaviour. The time delay in the guide example simply warrants the minimum separation stipulated between successive sporadic activations. The downside of the approach, however, is that the descriptive model of the task submitted to static scheduling analysis would typically omit to represent the regressive behaviour (the periodic one in this particular case), thereby discounting the consequent overhead incurred upon execution of the time suspension. Whether this inaccuracy of representation may be acceptable or else may be deemed to detract assurance value from the verification process remains to be seen, while it is likely to depend on domain-specific practices and requirements.

Another approach (which was the one the GOCE developer was compelled to by the analysis tools) is to rigorously comply with the requirement for tasks to only allow a single suspension point, so as to permit a uniform characterisation of their concurrent behaviour as well as *absolute iden-*

tity among design, code and model representations. As observed in [10], this approach promotes rigid specialisation of tasks, with consequent increase in density of the tasking population of the system. Indeed, the GOCE PASW developer reports this increase, but expects it to have negligible space and time impact thanks to the light-weight nature of the Ravenscar runtime. In addition to arguably valuable rigour of discipline, this approach has the virtue (and perhaps the penalty, as some designers may claim) of unifying control flow — the suspension points — with data flow — the interfaces involving suspension —. This unification spreads real-time requirements on individual control actions over a range of collaborating tasks interconnected by internal data and control flows. Handling this unification effectively is not immediate and it may well require specific design effort. The GOCE PASW developer reports with some disappointment that the excess of effort incurred did not produce, in their view, a cleaner and leaner design.

Commentary. The author's opinion on this issue is that at least two key ingredients are necessary for this approach to produce actual benefits:

- a set of *design patterns* or even specific *code frameworks* (with the distinction between the two as drawn by [4]) that show how multiple suspension points can be mapped onto an interconnected group of single-suspension-point tasks;
- guidance on the way real-time requirements and attributes (including offset, which assumes particular importance under this scheme) should be set on individual tasks and on how end-to-end response time analysis can be performed for the whole group of such tasks.

It is probably not the case that either of the above two alternative design approaches would equally suit all needs and situations. It may then well be useful that the Ravenscar Guide [1] discusses the merits of both alternatives and provides some guidance on how to compare them and how to adopt either one effectively.

3.3. Archetypal Design Patterns: Multi-Task Processes and Reified I/O Actions

The GOCE PASW development provides express confirmation of the "creeping demand for design patterns" that arises from rigid adherence to requirements R4-5.

The first element of the emerging design pattern is the split of the OBRTC service task into a pair of single-suspension-point tasks: a "starter" task, which initiates the OBRTC read-out calling a non-suspending protected procedure; and a "continuer" task, which undertakes to complete the job, once released from the closed protected entry which sanctions the availability of the OBRTC read-out and which is opened by the interrupt procedure.

The "start" task can freely be made sporadic or periodic as the application requires. The "continuer" task instead is forced to be sporadic by the nature of the I/O channel interface.

This archetypal design pattern occurs at various places throughout the GOCE PASW architecture. The developer however found it difficult to transpose the high-level deadline on the single logical entity collectively in charge of the transaction (and called *process* in GOCE) into finer-grained deadlines, activation attributes and precedence relationships on the set of actual tasks that implement it.

Problem Outline. This archetypal design pattern needs further sophistication to fully meet requirements R1-3.

As we noted earlier, the same I/O channel used for the acquisition of OBRTC read-out is shared for several other hardware devices among multiple *processes*. This situation gives rise to the following further requirements:

R6. the channel can only perform I/O operations in strict sequential order, one at a time, from start to completion

which implies mutual exclusion on access to the channel resources to protect the execution of individual operations from possible interference; and

R7. each process that shares the channel must post its request for a specific I/O operation to a multiplexer service and then wait for its required operation to complete

where full Ada would model such a channel multiplexer out of a protected object equipped with:

- one protected entry per allowable I/O operation request, each with a barrier that turns true when the channel is idle *and* a new operation can be started;
- one protected entry that waits for the current operation to complete and then opens the channel-idle part of the above barriers;
- one protected interrupt procedure that, on the operation completion signal, opens the other part of the above barriers.

The Ravenscar Profile only allows protected objects with at most one entry. Requirement R5 further requires that no more than one task ever queues on that entry. Some elaborate circumlocution must therefore be used to model the GOCE multiplexer service. An important constraint on the allowable circumlocution, though, is that it must cause neither of the tasks in the "starter-continuer" design pattern to

incur any further suspension point other than those explicitly intended for them.

The GOCE solution promotes a higher-level composite design pattern, which actually *reifies* I/O actions.

As the I/O operations must be serialised, the "starter" task of any relevant process must post its operation request into a protected queue and must do so in a way that incurs no suspension. This is easy to warrant, for the maximum number of incoming requests is as statically bounded as the number of client processes in the system, so that the size of the queue can be set accordingly. A dedicated sporadic task will fetch individual requests from the queue by calling on a protected entry with a barrier that stays open as long as requests are enqueued *and* the channel can accept new requests, and will subsequently initiate the corresponding operation on the channel.

The "continuer" tasks of the involved processes must not all queue on the one and the same protected entry whose barrier signals the the completion of the current operation and which delivers the result of it. Each "continuer" task shall then have to wait on an *own* protected synchronisation object whose address will have to be tagged to the operation request issued by the corresponding "starter" task. Each such synchronisation object will have a protected entry for the "continuer" task to wait on, and a signaling protected procedure that sets the entry barrier to true when the corresponding operation has completed. The signaling procedure will be invoked by the interrupt protected procedure attached to the I/O channel hardware interrupt to deliver the operation result, for which purpose it shall therefore have (shared) access to the current operation descriptor.

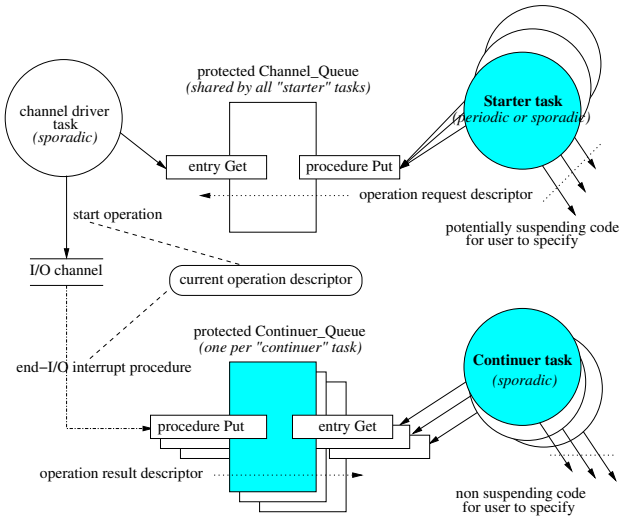

Figure 1. Component diagram of the "reified I/O actions" design pattern.

```
protected type Continuer_Queue
  (Prio : System.Priority) is
   procedure Put(Result : in ...);
   entry Get(Result : out ...);
   pragma Priority(Prio);
private
   Last_Result : ... := ...; -- holds only the last operation result
   Data_Ready : Boolean := False;
end Continuer_Queue;
type CQ is access all Continuer_Queue;
type Op_Req is record
   -- whatever necessary
end record;
type Op_Req_Queue is array(Positive range <>) of Op_Req;
protected type Starter_Queue
  (Prio      : System.Priority;
   Contents : Positive) is
   procedure Put(Op : in Op_Req);
   entry Get(Op : out Op_Req);
   pragma Priority(Prio);
private
   Buffer    : Op_Req_Queue(1. .Contents);
   Allowable : Boolean := False; -- the barrier condition
end Starter_Queue;
type SQ is access all Starter_Queue;
type SQ_P is access protected procedure(Op : in Op_Req);
type Op_Descr is record
   Op        : Op_Type;
   Sync_Obj : CQ; -- the Continuer_Queue to call
end record;
type Op_Descr_Queue is array(Positive range <>) of Op_Descr;
protected type Channel_Queue
  (Prio : System.Priority;
   Contents : Positive) is
   procedure Put(Op : in Op_Descr);
   entry Get(Op : out Op_Descr);
   pragma Priority(Prio);
private
   Buffer : Op_Descr_Queue(1. .Contents);
   Data_Ready : Boolean := False; -- the barrier condition
end Channel_Queue;
type IOCQ is access all Channel_Queue;
type IOCQ_P is access protected procedure(Op : in Op_Descr);
task type Starter -- sporadic
  (Prio             : System.Priority;
   Susp_Op      : SQ; -- to carry the activation event
   Sync_Obj     : CQ; -- to include in Op_Descr objects
   Non_Susp_Op : IOCQ_P) is -- to post Op_Descr objects
   pragma Priority(Prio);
end Starter;
task type Continuer -- sporadic
  (Prio             : System.Priority;
   Sync_Obj     : CQ; -- to carry the activation event
   Non_Susp_Op : DUQ_P) is -- to notify a user
   pragma Priority(Prio);
end Continuer;
```

Figure 2. Model code of the pattern components (specification).

This design pattern is named after the need for I/O operation requests and the corresponding results to be reified into data objects (descriptors) for client and server objects to manipulate. Figure 1 depicts the components of this archetypal design pattern. Figure 2 shows the model code for the pattern, in which we rendered the "starter" task sporadic.

In the developer's opinion this design pattern complicates static analysis in two ways. Firstly, the time offset that separates the release of the "starter" task from that of the "continuer" task in the same process depends on the number and the nature of the enqueued I/O operations, which may be difficult to bound without incurring excessive pessimism. Secondly, in order for the pattern not to incur undesirable architectural coupling, the "end-IO interrupt procedure" shown in figure 1, should use an access-to-protected value (stored in the current operation descriptor) to call the protected signal procedure for the completed operation, exactly in the way shown in figure 3, which makes the corresponding call graph harder to construct for a worst-case execution time analysis tool like Bound-T [8], which was used in GOCE.

The developer also comments that the Profile requirement for single-entry protected objects was felt as an obstacle to expressing the same level of execution pacing between producer and consumer tasks nicely attained by bounded buffers implemented as multi-entry protected queues.

Commentary. The author views the emerging pattern as very attractive and regards the two above complicating factors as tractable. The first one could be easily alleviated by the addition of specific guidance in the Ravenscar Guide [1], thereby also adding to the usefulness and productivity of the Profile. The second one arguably concerns support technology: its resolution would require specific (yet generally desirable) enhancements to the call-graph constructor component of Bound-T, capable of treating task bodies such as those shown in figure 3 in which the procedural actions are denoted by access values and the reified I/O actions (of type Op_Descr in our model code) contain the pointer to the protected object to call to release the continuer task.

As for single-entry protected objects, example 12 of the Ravenscar Guide shows how a bounded buffer can be coded without violating the Profile restrictions. Perhaps the GOCE developer could have adopted this solution to achieve execution pacing. Yet, its use would have caused the "starter" task in the design pattern shown in figure 1 to engage its single suspension point with the call to procedure Put, which would now become potentially suspending in spite of its innocent-looking specification, thereby becoming less attractive, perhaps even unfit, as a design pattern for high-integrity systems.

```
task body Starter is
   Req    : Op_Req;
   Req_Op : Op_Descr;
begin
   loop                                                          5
      -- suspending operation
      Susp_Op.Get(Req);
      Req_Op.Op_Type := Req.Op_Type;
      Req_Op.Sync_Obj := Sync_Obj;
      -- non-suspending operation                               10
      Non_Susp_Op.all(Req_Op);
   end loop;
end Starter;
task body Continuer is
   Result : ...;                                                 15
begin
   loop
      -- suspending operation
      Sync_Obj.Get(Result);
      -- non-suspending operation to propagate notification     20
      Non_Susp_Op.all(...);
   end loop;
end Continuer;
```

Figure 3. Model code of the task bodies of the starter-continuer process.

3.4. Further Design Patterns: Paced Data Pipeline

Problem Outline. The developer views a further design pattern emerge from the data-flow network that interconnects data sources and data sinks in the GOCE system. The sought pattern should cater for the pacing of the data pipeline from a (command or data) source to a data sink that operate asynchronously, in a manner that satisfies requirement R3. The activity of the pipeline-pacer *process* is exposed to two logical suspension points: command or data ready at the source; and sink ready to receive. As source and sink are asynchronous, some buffering will be needed at any of the two ends to handle upstream or downstream congestions. Requirement R3 suggests that the buffer should be placed at the end that requires the lowest size.

One particular instance of this problem in GOCE comprises: one command source, which carries requests for specific data transfers; one data sink, destination of the required transfer; and one remote data storage, from which the data have to be first acquired before being transferred to the sink. An I/O channel connects the main processor to the remote data storage from which the required data have to be fetched. The presence of the I/O channel implies that the emerging design pattern could conveniently build on top of the one outlined in section 3.3, with the "starter" task at the upstream end and the "continuer" task at the downstream end of the pipeline.

As incoming commands are smaller than transfer units,

buffering on input to the "starter" task comes cheaper. This task shall therefore fetch for execution from its input queue one command at a time, if available, as soon as the data sink has declared sufficient capacity for the relevant data transfer. These two suspension conditions must be logically "and-ed" at any status change into the simple barrier value allowed by the Ravenscar Profile. This requirement implies a protected command queue with a non-suspending procedure Put for the insertion of both commands and notifications of sink ready, and an entry Get with a barrier condition that is the logical and of the two activation conditions. (In fact, other pattern designers might prefer sharper specialisation of protected operation and therefore require that Put only be used for non-suspending command insertion and a distinct Release procedure be used for non-suspending notification of sink ready. There is value in both alternatives. The author has attached greater value to using protected objects with the standard Put and Get interface, viewed as obvious instances of a very basic Ravenscar design pattern.)

When a new operation request has arrived and the downstream user is ready for new data, the "starter" task is released off the entry Get queue and may initiate the required operation. As a result of that the transfer data unit is copied from the remote storage into a single-position buffer owned by the data sink driver task. At the downstream end of the paced pipeline *process*, the "continuer" task simply waits for the arrival of the I/O channel interrupt notifying completion of the relevant operation to inform the data sink driver task that the required transfer unit is ready. This effect we easily obtain by having the access-to-protected-procedure discriminant parameter of the "continuer" task type shown on line 55 of figure 2 point to the non-suspending notification operation of the user queue.

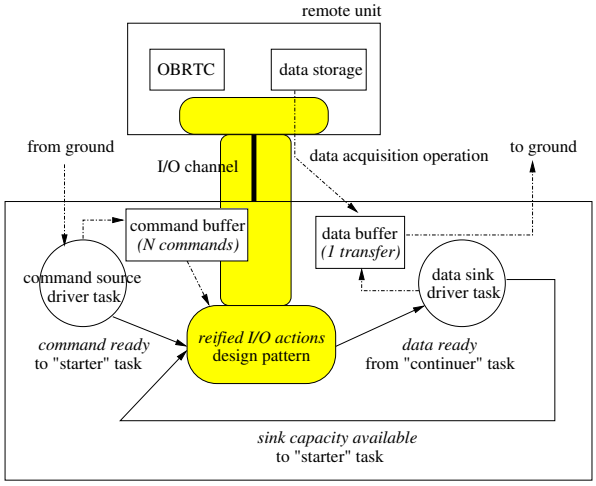

Figure 4. Component diagram of the paced pipeline architecture.

The resulting design pattern is outlined in figure 4, while figure 5 shows how to adapt the earlier model code of the Starter task to the needs of this higher-level pattern.

```
-- for addition in the private part of Starter_Queue
   Release_Quotum : Natural := 1; -- downstream condition
   Contains : Natural := 0; -- upstream condition
   -- the barrier condition now becomes:
   -- (Contains > 0) and (Release_Quotum > 0);
-- for adaption of the Op_Req type
type Source is (Upstream, Downstream);
type Op_Req is record
   Src      : Source; -- and whatever else necessary
end record;
protected body Starter_Queue is
   procedure Put(Op : in Op_Req) is
   begin
     case Op.Src is
       when Upstream =>
         ... -- the upstream client posts its request
         Contains := Contains + 1;
         if Contains > Contents then
           -- errror management
         end if;
       when Downstream =>
         -- the downstream user notifies its readiness
         Release_Quotum := Release_Quotum + 1;
     end case;
     Allowable := (Contains > 0) and (Release_Quotum > 0);
   end Put;
   entry Get(Op : out Op_Req) when Allowable is
   begin
     ... -- fetch one request
     Contains := Contains - 1;
     Release_Quotum := Release_Quotum - 1;
     Allowable := (Contains > 0) and (Release_Quotum > 0);
   end Get;
end Starter_Queue;
```

Figure 5. Model code adaptions to the Starter task with double use of the single Put procedure.

The developer's view is that the single-suspension-point requirement complicates the use of this design pattern, making it difficult to determine how the two suspension points above (command ready and sink ready) give rise to the single minimum interval attribute required by scheduling analysis for the "starter" task of the paced-pipeline *process*.

Commentary. The author regards the design pattern outlined in figures 4 and 5 to readily match a range of real-life situations that occur in applications like GOCE. (Figure 6 shows an instance of use of the pattern model code to represent two concurrent "starter-continuer" processes that share the same I/O channel.)

In order to win the trust of the designer, any design pattern has to produce quantifiably superior benefits to the

inevitable pain of frustrating the designer's creativity and freedom. In the author's opinion what such patterns still miss to gain wider acceptance is comprehensive guidance on the process of conducting static analysis on them (response time analysis above all). Once again, the Ravenscar Guide [1] seems to be the most natural place for it.

```
--2 Starter queues
SQ_1        : aliased Starter_Queue(12,5);
SQ_1_Ptr    : constant SQ  := SQ_1'Access;
SQP_1_Ptr : constant SQ_P := SQ_1.Put'Access;
--+
SQ_2        : aliased Starter_Queue(8,5);
SQ_2_Ptr    : constant PSQ := SQ_2'Access;
SQP_2_Ptr : constant PSQ_P := SQ_2.Put'Access;
--2 Continuer queues
CQ_1        : aliased Continuer_Queue(15);
CQ_1_Ptr    : constant CQ  := CQ_1'Access;
--+
CQ_2        : aliased Continuer_Queue(15);
CQ_2_Ptr    : constant CQ  := CQ_2'Access;
--1 shared Channel queue
IQ        : aliased Channel_Queue(16,2);
IQ_Ptr    : constant IOCQ := IQ'Access;
IQP_Ptr : constant IOCQ_P := IQ.Put'Access;
--2 Data user (downstream) queues
DQ_1        : aliased Data_User_Queue(13);
DQ_1_Ptr    : constant DUQ := DQ_1'Access;
DQP_1_Ptr : constant DUQ_P := DQ_1.Put'Access;
--+
DQ_2        : aliased Data_User_Queue(9);
DQ_2_Ptr    : constant DUQ := DQ_2'Access;
DQP_2_Ptr : constant F.DUQ_P := DQ_2.Put'Access;
--2 Starter tasks
PS_1 : Starter(10, SQ_1_Ptr, CQ_1_Ptr, IQP_Ptr);
PS_2 : Starter(6, SQ_2_Ptr, CQ_2_Ptr, IQP_Ptr);
--2 Continuer tasks
PC_1 : Continuer(11, CQ_1_Ptr, DQP_1_Ptr);
PC_2 : Continuer(7, CQ_2_Ptr, DQP_2_Ptr);
```

Figure 6. Example instantiation of the pattern code with 2 "starter-continuer" processes.

4. Conclusion

Requirement R4, which adds to the Ravenscar Profile the restriction for tasks to only have a single suspension point, was found to limit procedural abstraction of suspending operations.

Our response to this observation was two-fold.

On the one hand, we argued that the additional, stricter requirement directly follows from turning adherence to the Ravenscar Profile into a full-blown design methodology, whose added value surely is no less than the perceived loss of design abstraction.

On the other hand, we noted that the Ravenscar Profile enforces a design discipline that cannot tolerate the concealment of potentially suspending operations in lower-level service code. The net consequence of this prohibition is that procedural abstraction may not be allowed to hide actions that may effect on the concurrent and timing behaviour of the caller. When safety of design is concerned, the benefit of this prohibition prevails over any frustration that may ensue from adherence to it.

Requirement R4 was also found to force task specialisation and consequently increase the task population in the system, thereby rising the memory size required for task stacks (which the programmer can still control, though). The GOCE designer further feared that this requirement might also cause an increase in the runtime overhead components that are linear in the number of tasks, but, as a matter of fact, very few ones actually seem to be, as can be inferred from [11].

Requirement R5, which only allows a single call for each protected entry, does cause an obvious increase in the number of protected objects, which is however anticipated to have negligible space and time impact. (Actual measurements in GOCE should soon corroborate this claim.)

Of greater momentum was the designer's complaint that the resulting tasking profile was perceived to place severe limits on the way control flow can be expressed within a task. The paradigm shift here is that a single algorithm can very likely no longer be embedded in the control flow of a single task, but the key elements of it must now spread over several protected data structures and state variables (in essence, over a *process*). The developer's argument was that this phenomenon makes the algorithms more difficult to express and to follow.

In fact, this difficulty seems to be inherent with any significant shift in the design paradigm that follows from the adoption of a rigid discipline. The promotion and enhancement of design patterns that embed recurrent algorithms seem to be an attractive solution to the problem.

Splitting tasks implies splitting the apportionment of real-time requirements and attributes that, at system level, are naturally expressed in terms of a single logical activity. Better guidance may be required for the user to effectively carry out the resulting end-to-end response time analysis.

Overall, the bottom line seems to be how conduciveness to static analysis stands against expressive power. In the author's opinion, developers should accept that several aspects of the full language model (and not just in Ada) simply exceed the power of current static analysis techniques. If the latter is the prevailing requirement, then some element of expressive power must be given up for a better good.

The Ravenscar Profile cannot and should not compete for expressive power with the full language. This suggests once more that the availability of a range of design patterns may help the user appreciate how they can solve recurrent problems with greater emphasis on the tamed expressive power

of the Profile than on the severity of its constraints.

Acknowledgments. This work was supported, in part, by the COFIN 2002 project no. 2002017471 (COVER) and the FIRB 2001 project "Abstract Interpretation and Model Checking for the Verification of Embedded Systems", both funded by the Italian Ministry for Education and Research.

The author gratefully acknowledges the permission by SSF, the developer of the GOCE PASW, to use the yet unpublished contents of [5] and the value of various private conversations with their engineers, especially Niklas Holsti, interleaved with the production of the cited paper, as the basis to the reflections presented in this paper.

References

[1] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. Technical Report YCS-2003-348, University of York, 2003. `http://www.cs.york.ac.uk/ftpdir/reports/YCS-2003-348.pdf`.

[2] 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.

[3] 2nd Int'l Workshop on Worst-Case Execution Time Analysis. `http://www.cs.york.ac.uk/rts/wcet2002`, June 2002. Euromicro.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissideds. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1999.

[5] N. Holsti and T. Langbacka. Impact of a Restricted Tasking Profile: The Case of the GOCE Platform Application Software. In J.-P. Rosen and A. Strohmeier, editors, *Reliable Software Technologies — Ada Europe 2003*, LNCS 2655, pages 92–101. Springer-Verlag, June 2003.

[6] S. Mazzini, M. D'Alessandro, M. Di Natale, A. Dominici, G. Lipari, and T. Vardanega. HRT-UML: Taking HRT-HOOD onto UML. In J.P. Rosen and A. Strohmeier, editors, *Proceedings 8th International Conference on Reliable Software Technologies — Ada Europe 2003*, LNCS 2655, pages 405–416. Springer-Verlag, June 2003.

[7] S. Mazzini, M. D'Alessandro, M. Di Natale, G. Lipari, and T. Vardanega. Issues in Mapping HRT-HOOD to UML. In G. Buttazzo, editor, *Proceedings 15th Euromicro Conference on Real-Time Systems*, pages 221–228. IEEE, July 2003.

[8] Bound-T Execution Time Analyzer, Home Page. `http://www.bound-t.com`, June 2003.

[9] T. Vardanega. Development of On-Board Embedded Real-Time Systems: An Engineering Approach. Technical Report ESA STR-260, European Space Agency, 1999. ISBN 90-9092-334-2.

[10] T. Vardanega and G. Caspersen. Engineering Reuse for On-board Embedded Real-Time Systems. *Software - Practice and Experience*, 32(3):233–264, 2002. John Wiley & Sons.

[11] T. Vardanega, J. Zamorano, and J. de la Puente. On the Dynamic Semantics and the Timing Behaviour of Ravenscar Kernels. *Real-Time Systems*, 2003. To appear.