Engineering software reuse for on-board embedded real-time systems

Tullio Vardanega¹, a Gert Caspersen²

¹ European Space Research and Technology Centre, Keplerlaan 1, 2200 AG Noordwijk, NL
² Terma, Bregnerodvej 144, 3460 Birkerod, DK

SUMMARY

The dimensions of concern to ambitious reuse initiatives largely exceed the provision of desired functionalities. A domain-specific blend of functional, technical and quality considerations determines the reuse potential of software reuse assets. In a recent project we realised considerable reuse dividends from coupling the definition of a reference software architecture and of an associated set of standard interfaces with the selection of a powerful computational model equipped with reuse-geared enabling technology. In this paper we present our engineering approach to the project and show how it earned us a good balance between the preservation of predictability and scalability and the maximisation of the reuse objective.

KEY WORDS: Software reuse; reuse metrics; concurrency; Ravenscar profile; generic instantiation; object model.

INTRODUCTION

The consolidation of the engineering research into software reuse (cf. e.g.: [26, 25]) has recently penetrated the space business and prompted the question whether and how can reuse lower the rising development cost of new-generation software-intensive on-board systems.

The question is relevant for at least two good reasons. On the one hand, environment characteristics and mission requirements curtail the productivity of space software projects for on-board application in comparison to other business sectors (cf. e.g.: [24]) and make them ill-equipped to meet increasing user demands without incurring serious cost overruns. On the other hand, the only practical opportunity for the development of software systems for on-board application to respond to the new business climate, which goes under the name of ‘cheaper, faster, better’, will be to attain a considerable productivity
increase. By way of example, earlier work [30] anticipated the software productivity demand to quadruple in the next few years, for twice as big software to be delivered in half the time, within no less than the present envelope of dependability.

Software reuse is a useful response to the challenge. It may increase the dependability of the product via defect removal from the reuse baseline. It may also increase productivity by way of work avoidance. Work avoidance alone however is only a contributing factor to the achievement of the overall productivity objective [4]. Other complementary process means and engineering techniques have to contribute to it, too. This cautionary observation is necessary to stop unrealistic expectations (and, thus, unjustified disappointments) being placed on the ultimate yield of reuse.

In this paper we discuss the engineering characteristics of a software reuse framework specifically designed for new-generation on-board systems. We first lay down the methodological foundation to our work. We then illustrate the strategic, architectural and technological choices that most characterise our development. Subsequently, we examine a representative cross section of our implementation approach. Finally, we evaluate the performance of our software product with respect to predictability and sizing, which are crucial concerns to our application domain, and reuse ratio, which is the chief objective of our initiative.

BACKGROUND

Arguably, the definition and the enforcement of standard specifications is a most effective means to facilitate effective reuse within and across industry.

A standard specification named Packet Utilisation Standard [18] (PUS for short) has recently emerged at the European Space Agency to address a significant proportion of the recurrent data handling and control functions on board their satellite systems. Although originally intended to prescribe the application-level protocol for ground to on-board communications, the standard effectively embodies a vast amount of reuse potential for the on-board system also; primarily by way of promoting a domain-specific descriptive model and a domain-specific architecture of the system (in the way advocated, for example, by [19]). The PUS definitions form the specific domain of the reuse initiative discussed in this paper.

A serious software reuse initiative needs an effective software development process to support it. Earlier work [29, 30] argued that an iterative and incremental development process is better equipped than the traditional ‘waterfall’ model to cope with the increasing complexity of the on-board software product and the concurrent reduction of the development schedule. The process vision we subscribe to calls for a software architecture scalable enough to accommodate successive increments and also capable of assuring the integrity of the system and the convergence of the process.

Three major ingredients are essential facilitators to a reuse-centric realisation of this process of ours: (a) an application model that favours the scalability of the system and promotes a reference architecture, which leads to ‘reuse by architecture’; (b) a software reuse framework that accelerates the construction of systems in accord with the application model and its reference architecture, which allows ‘reuse by artifacts’ and requires attention for both the static and the dynamic aspects of the system; (c) enabling technology that supports the production of reusable, reliable and predictable software, which permits ‘reuse by technology’.
To seize the opportunity presented by the emergence of the PUS, the European Space Agency funded the development of OBOSS (short for: on-board operations support software) [27], a library of reusable software components along with a domain-specific architecture for their utilisation, designed for the construction of on-board data handling systems compliant with the standard.

The programmatic objective of OBOSS is to constitute a vehicle for inter-company ‘open’ software reuse as well as a model for intra-company ‘proprietary’ reuse.

The OBOSS concept effectively builds upon specific instances of the process facilitators mentioned above. The OBOSS notion of software reuse in fact encompasses:

- a reference architecture that matches the PUS application model (corresponding to reuse by architecture)
- a set of software components with standard interfaces and prescribed operation semantics, organised in what we call a software reuse framework (corresponding to the static facet of reuse by artifacts)
- a predictable and scalable computational model (corresponding to the dynamic facet of reuse by artifacts)
- a specific reuse strategy that optimally uses the selected language support (corresponding to reuse by technology).

We will return to each of these aspects in the following section and later in the paper.

A CASE FOR SOFTWARE REUSE

Application Model

Satellite systems typically perform their operation under the supervision of a control centre based on ground. The ground centre exercises its authority on the on-board system by the issue of telecommands (TC) and the reception of status and verification information contained in the response telemetries (TM) returned by the satellite. The servicing of the TC/TM flow between the ground centre and the orbiting satellite constitutes one fundamental function of the real-time software embedded in the spacecraft controller, the single platform component that integrates together all of the control and service functions performed on board. The set of communication-related activities associated with the servicing of the TC/TM flow is collectively termed Data Handling and Control (DHC) system.

The European Space Agency’s Packet Telemetry and Packet Telecommand Standards [17, 16] and the CCSDS Recommendations from which they are derived [9, 10] address the end-to-end transport of TC and TM data between user applications on ground and the respective destination and sources on board and, accordingly, prescribe the general layout of TC and TM packets. The PUS [18] complements and extends those standards by defining a packet-based protocol for the transmission and on-board distribution of TC, and a standard set of monitoring and control services to be performed on board. The PUS defines those standards by prescribing the service model, i.e., the manner in which service providers are to behave on the arrival of service requests, and the inner structure of the TC and TM packets carrying respectively service requests and the corresponding service reports.

The purpose of the standard was to cover the known requirements of present and future space missions while staying open to the introduction of additional services and service capabilities as the
need arises. One interesting corollary of the openness of the standard is that the vast majority of individual services are defined independently of one another, so that the execution of any one of them would not involve the execution of any other. The notion of service capability denotes the possible use of the service as prescribed by the standard, which may entail mandatory and optional elements. The definition of the PUS is thus the product of a very comprehensive domain analysis performed over a broad spectrum of missions and, consequently, also an especially suitable basis for the construction of software reuse libraries for on-board and on-ground utilisation.

In the context of the PUS, the on-board entities that are capable of receiving TC packets and of generating TM packets are referred to as ‘application processes’. From the standpoint of the ground operator, therefore, application processes are service providers and specific entities of the ground centre(s) are service users. A natural extension to this model, however, allows application processes to also issue service requests, which the ground issues encoded as TC packets. Application processes would in this case become true peers to service users on ground and therefore interchangeable with them. This extension entails simple modifications to the outer structure of standard TC and TM packets and is essential to autonomous systems where the supervisory capabilities traditionally placed on ground are delegated to application processes on board.

An application process is uniquely and statically identified for the entire lifetime of a mission. No restrictions are placed on the mapping between application processes and the usual functional subdivision of a satellite into subsystems and payloads. Accordingly, application processes may operate as centralised providers of monitoring and control services or as dispatching agents for remote subsystems equipped with own processing capabilities.

As a by-product of prescribing the protocol for the transmission and distribution of TC and the associated service model, the PUS evokes the notion of a PUS-based on-board architecture, and a particularly loosely-coupled one at that. The fact that TC carry service requests that are directed to application processes determines an implicit relationship of containment between application processes and a corresponding set of service capabilities. An application process may embed any number of service capabilities, in the nature and quantity required to operate on the section of the space segment specifically assigned to it. The loosely-coupled nature of the PUS on-board software architecture directly follows from this relationship of containment: a service request addressed to one application process does not perturb the activity (or the quiescence) of any other application process.

Figure 1 depicts the top-level architecture that emanates from this interpretation of the PUS. The figure maps the abstract view of a PUS-based space system onto the traditional model. The components of the former are, at times, service users and / or service providers. In the latter, instead, application processes are service providers, ground centres are service users, TC embed service requests and TM embed service reports.

The notions entailed by the PUS model extend the potential coverage of a software reuse initiative from a scattered collection of utilities and functions to a library of architectural elements and service modules.

**Reference Architecture**

Several key features of a PUS-compliant DHC software architecture emanate directly from the conceptual model shown in figure 1. First and foremost, the bulk of the software infrastructure in charge of on-board communications that adhere to the standard may stay largely the same across missions,
with regard to both the structure and the interpretation of packets. Furthermore, the loosely-coupled
character of the PUS architecture and the event-driven nature of the associated service model (with the
event being the arrival of a service request) imply that the processing of independent commands may
proceed in parallel throughout the system as a whole as well as within individual application processes.

Consequently, the software architecture of application processes themselves naturally break down to
two major components: a recurrent infrastructure for the support of the routing and the interpretation
of service requests and service reports within and among application processes; and an application-
specific collection of standard service capabilities to be embedded within the individual application
processes.

OBOSS provides configurable support for packet-based message passing between the ground
segment and on-board application processes and for the servicing of the relevant commands. The
system architecture promoted by OBOSS proceeds directly from the reference model depicted in
figure 1 and, as its model, it is loosely-coupled and inherently scalable. Control flow within OBOSS is
predominantly event-driven, with an event corresponding to the arrival of a packet-encoded message
from any source in the system, whether ground or on-board based.

Two forms of communications take place within a PUS system, and thus within OBOSS. The
communication between application processes is packet based. The communication within application
processes is also packet based as long as it involves a local destination, while it relies on user-supported
protocols for all communications that involve remote destinations. Remote destinations that feature
own control processing capabilities are termed subsystems. Remote destinations that do not possess
those capabilities are called units.

The packet-based communication architecture within OBOSS is centred around a Packet Router that
distributes packet-encoded messages to the mailbox of the destination application processes.

OBOSS provides each application process with a local Packet Dispatcher that fetches the packet
from the local mailbox and delivers it to the service capability concerned.

OBOSS allows DHC-based application processes to operate as communication agents for on-board
subsystems that are physically disjoint from the DHC. In that case, the application process resident on
the DHC embeds no service capability and simply forwards packets to and from the remote subsystem,
which operates as the actual service provider (or service user). OBOSS preserves the locality of its packet-based communication infrastructure by providing fully reusable components to interface agents to the Packet Router.

The most obvious instance of OBOSS agent represents the ground centre that operates the spacecraft. Another typical instance of OBOSS agent occurs in systems comprising payload instruments equipped with their own control computers. The OBOSS agent in this case would operate as the local communication proxy for the remote control unit of the corresponding payload instrument. The instrument control unit itself would in this case embed the whole service capability set of the application process. Application Process \( n \) in figure 1 and in figure 2 illustrates this notion by comprising an OBOSS agent on the DHC platform and a service capability set on the remote subsystem.

Figure 2 depicts the main architectural components of a software system based on OBOSS. (The reader should notice the structural match of the OBOSS architecture in figure 2 with the abstract system model shown on the right side of figure 1.)

The box tagged ‘On-board Traffic Management’ in figure 1 logically corresponds to the boxes tagged ‘System Infrastructure’ and ‘Platform Services’ in figure 2. The System Infrastructure component carries out the hardware-independent packet handling and the Platform Services component encapsulates all dependencies from the specific physical communication network on board.
The OBOSS reuse framework provides a catalog of software components that fully support the system infrastructure. User-supplied components that conform to the OBOSS static and dynamic interfaces are instead required to complete the implementation of the platform services.

Software Reuse Framework

The OBOSS reuse model enforces the adoption of the reference architecture depicted in figure 2 and of the communications infrastructure associated with it. To this end, the OBOSS reuse assets comprise: (i) a set of generic units for constructing application processes; (ii) a set of generic units embedding service capabilities that cover a subset of those specified in the PUS; (iii) a vast amount of guidance information for the construction of mission-specific DHC systems.

Two subsequent stages of domain analysis (cf. [13] and [14]) determined the core of PUS services most suited for software reuse. Two key criteria guided the selection of services and service capabilities: (i) the likelihood of use in future missions, whereby a service frequently used would constitute a natural reuse candidate; (ii) the cost of instantiation for a mission, whereby a service specification that allows for a design with low instantiation cost (i.e., that is with significantly more recurrent code than what it would need to make an instantiation) would be a valuable candidate for reuse.

Ref. [18] defines 17 standard service types, each uniquely identified by an index in the range [1..17]. Domain analysis identified 9 of those standard services for OBOSS to support. Of those, 2 correspond to the TC verification service and the event reporting service, which are intrinsic (and compulsory) to the very essence of PUS application process. All other 15 standard services are optional in the sense that a PUS system may provide any subset and any combination of them.

Overall, the standard allows for a total of 256 service type identifiers. Those in the range [18..127] are reserved for future additions to the standard set. Those in the range [128..256] are free for the user to define in an application-specific fashion.

The view held by the authors of the cited domain analysis is that the OBOSS selection of service capabilities covers up to 90% of the typical PUS coverage of new-generation on-board systems.

The notion of generic unit constitutes the building block to the OBOSS architecture. In language-neutral terms, a generic unit is a template that factors out commonalities in the treatment of abstract actions or data structures, which are associated to generic formal parameters. Correspondingly, a generic instantiation is a copy of the generic unit that provides a concrete definition for those actions and data structures, in the form of generic actual parameters.

A specific concurrent behaviour may be associated to the execution of a generic instantiation simply by (logically) embedding the generic instantiation defining that behaviour within the corresponding generic unit. In this way, the concurrent behaviour of an outer generic instantiation in OBOSS becomes a permanent attribute of the corresponding generic unit. The inner unit that determines that behaviour is not under the direct control of the user, who can only define selected configuration parameters of the embedded concurrent unit, such as execution priority and stack size. As a reflection of this strategy, the core of an OBOSS-based system is constructed by assembling together a battery of generic instantiations, a collection of adapted templates and a set of interface procedures.

Figure 3 explodes the view of the OBOSS internal architecture by zooming into the inner structure of an application process $i$ embedding an instance of PUS service capability $j$. In keeping with its approach to reuse, OBOSS implements PUS service capabilities as a hierarchical collection of generic
units. The polygon tagged ‘Service $ij$’ in figure 3 represents the outermost generic instance for service $j$ in application process $i$.

Parameter configuration is the primary means offered by OBOSS to control how reuse assets group together into a coherent system architecture. As highlighted in figure 3, the OBOSS parameters belong in three main categories: ‘connectors’, which attach individual application components to the OBOSS system infrastructure; ‘attributes’, which capture specific functional characteristics of the individual component; and ‘terminals, which encapsulate dependencies on the physical architecture of the system in the OBOSS interface to user-supplied platform services.

‘Connector’ parameters interface the application process to the mailbox-based message passing system of OBOSS. The OBOSS reuse model provides templates for how the terminal interface requirements should be met. Those connectors effectively correspond to the user adaptation of a standard set of mailbox-based Send_Packet, Receive_Packet and Deposit template procedures, which OBOSS employs to transport packet-encoded messages to destination. The boxes tagged Packet Dispatcher and Packet Depositor in figure 3 represent the ‘connector’ parameters of application process $i$. Packet Dispatcher implements the outermost Send_Packet-based interface to the application process. Its implementation will reflect the nature and the topology of the service components embedded within the application process, each of which will provide the implementation of a Receive_Packet-based entry point. Packet Depositor deposits packet-encoded service reports produced by the application process in the mailbox of Packet Router, which forwards them to the intended destination.
‘Attribute’ parameters define the configurable characteristics of the specific application process or
the specific service incarnation within it (e.g.: unique identifier of the application process, service
policy, service priority). The component tagged ‘Application Process Parameters’ in figure 3 represents
the attributes emanating from the application process to the service incarnation in this instance.

‘Terminal’ parameters interface the application process to remote destinations that logically belong
to it. Those destinations are termed subsystems when they possess own control capabilities or else units
when they do not. In the former case, OBOSS only provides for the communication agent to the remote
subsystem and the ‘terminal’ parameter is the means that transports packet-encoded messages to and
from the agent. In the latter case, the ‘terminal’ parameter is used to enable the monitoring and control
capabilities embedded within the local service incarnations to acquire values from and send commands
to the relevant unit.

In order to maximise the symmetry of its user interfaces, OBOSS uses a pair of Send / Receive non-
suspending subprograms for the implementation of all its ‘terminal’ parameters. It is the responsibility
of the user to supply a non-suspending body that conforms to the OBOSS specification. The dashed box
tagged Platform Services in figure 3 globally represents all of the ‘terminals’ needed by an OBOSS-
based system, along with their interface counterpart within application processes. The component
tagged Bus Terminal in the same figure represents an instance of ‘terminal’ parameter that interfaces
the application process and its service incarnation(s) to the controlled remote unit via a given on-board
bus interface.

We shall return to these architectural notions later in this paper, when we illustrate the way OBOSS
implements a selection of them.

Enabling Technology

On-board systems are inherently concurrent. A large proportion of new-generation systems will be
even more so on account of the integration of an increasing number of control functions on a decreasing
number of processing nodes. Our need is, thus, for a computational model that facilitates the controlled
expression of that inherent concurrency.

Our choice originates from the definitions given in HRT-HOOD [7, 8] and relies on the principles
of fixed priority preemptive scheduling [2, 3]. HRT-HOOD extends the base HOOD design method
[21] in use at the European Space Agency by incorporating the abstractions supported by the revised
tasking model of Ada 95 [23] in the fashion prescribed by the Ravenscar Profile [5].

Our computational model allows for: periodic objects to model time-triggered threaded activities;
sporadic objects to model event-triggered threaded activities, for which the triggering event is either
a physical interrupt or a software synchronisation; and protected objects to model non-threaded
structures for data-oriented synchronisation.

The Ravenscar Profile represents the most compact and efficient set of language primitives needed
to implement our computational model. Architectures built in accordance with our computational
model and implemented using the Ravenscar Profile lend themselves to static timing analysis. Ref.
[30] describes a set of static analysis techniques that support this notion.

The Ravenscar Profile [5] defines a subset of the Ada 95 language that aims at predictable and
efficient use of concurrency in high-integrity real-time systems. The definition of the profile especially
seeks:
Staticness of task and protected objects, and of priority assignments throughout the system. The profile excludes both the use of such objects and types other than at library level and the use of dynamic priority assignments at run-time.

- Monotonic time and absolute delays for a more accurate generation of time-based events. The profile excludes the use of the coarse-grained Calendar package and of relative delays.

- Simplified mechanisms for inter-task communication centred on the use of asynchronous communication via protected objects, with the corresponding simplification of run-time behaviour. The profile excludes the use of rendez-vous based synchronous communication (hence task entries and select statements), asynchronous task control, abort statements and termination in general.

- Minimisation of blocking conditions by restricting protected objects to have at most one entry with a simple boolean barrier and a single caller.

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.

OBOSS faced two major challenges in the decision to shape its concurrency model in accord with the Ravenscar Profile. The first challenge was to express this concurrency in a manner that would fit in the OBOSS design philosophy. 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.

An interesting question indirectly raised by the OBOSS initiative is whether the Ravenscar Profile is sufficient for the construction of a complete on-board system of the new generation. This paper and associated earlier work (cf. e.g.: [28]) provide some, arguably solid, evidence to this effect.

As a direct consequences of these design decisions, OBOSS is entirely coded in Ada [22, 23] and is designed for direct execution on embedded Ada runtimes, so as to minimise dependence on external interfaces. No special requirements are placed on the implementation of the user-supplied components that are expected to co-exist with OBOSS in an operational system, except for those that directly interface with it, from which OBOSS requires static and dynamic conformance with a specific interface profile.

Two consolidated features of the Ada language proved an excellent match for distinguishing characteristics of the OBOSS architecture: (A) generic units as the main vehicle for software reuse; (B) embedded tasking in the way of the Ravenscar Profile for increased predictability and portability to embedded platforms.

The first implementation of OBOSS completed in 1997 [11], before mature Ada 95 technology, with its support for object orientation, was available for target platforms of interest to the project (cf. e.g.: [15]).

As a reflection of this origin, OBOSS is an Ada 95 system as regards the concurrency model, but a pure Ada 83 system in the respect of its sequential component. In keeping with the capabilities of Ada 83, the OBOSS objects have structure, state and operations but lack true inheritance.
(specialisation) and polymorphism, which are the distinguishing characters of full object orientation. In spite of this limitation and of the lack of an explicit ‘class’ keyword in the language, OBOSS objects are created by generic instantiation of base entities that we will call classes. OBOSS classes (and the corresponding objects) are made active by embedding the required type of concurrency within them.

The emergence of the Ravenscar Profile in 1997 [5] and of associated supporting products (e.g.: [12, 1]) permitted the OBOSS project team to revisit implementation choices made for its concurrency model. OBOSS 2nd generation [27] originated as the team’s response to these stimuli. All references to OBOSS in the remainder of this paper specifically denote OBOSS 2nd generation.

IMPLEMENTATION

Having looked into the design-level aspects of our engineering approach, we may now turn our attention to the concrete decisions we have made in the implementation of the OBOSS software reuse framework. In particular, we concentrate on the inner structure of the application process that we outlined earlier in figure 3. The application process represents the major component of an OBOSS system and it is the key to its intrinsic scalability.

The discussion that follows will also implicitly illustrate the way OBOSS enforces standard interfaces with prescribed operation semantics and the way this also caters for a predictable and scalable computational model.

Before going any further, however, we need to introduce the taxonomy of components that populate the OBOSS software reuse framework.

The OBOSS reusable software components belong in three main categories: (i) class objects, implemented as the instantiation of generic units; (ii) template objects, implemented as model library units, which the user must tailor (i.e., source edit) to suit the specific needs of the system; and (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. Class objects and auxiliary objects may give rise to 100% unmodified reuse, whereas template objects primarily serve as structural guidance to the implementation.

Passive objects at the top of an implementation hierarchy are composite (i.e., parent) objects the implementation of which needs child objects of passive type only. Active objects at the top of the hierarchy are composite objects that are implemented by at least one active terminal 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. Sporadic objects are further subdivided into interrupt sporadic and software sporadic depending on the nature of their triggering event. Cyclic terminal objects provide no methods for other objects to invoke, hence they only invoke the methods of others. Software sporadic terminal objects provide one single method (notionally named Start) which other

---

\(^1\) Recall that we use the notion of ‘class’ in a loose sense: an OBOSS class may be freely instantiated, thus becoming a (class) object, but it cannot be extended.
objects invoke to trigger the sporadic operation of the object. For interrupt sporadic terminal objects, the corresponding method is assumed attached to the corresponding interrupt source and accordingly invoked. The \texttt{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 \texttt{Start} have them implemented by child objects that are either passive or protected. Protected objects provide methods that warrant synchronous and mutually exclusive execution.

Figure 4 shows the HRT-HOOD graphical notations that we use to describe the internal architecture of an OBOSS system.

**OBOSS Units of Concurrency**

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 is to represent sporadic, cyclic and protected terminal objects as classes. In this manner, individual active terminal objects are simply obtained by generic instantiation of the corresponding class. To illustrate the approach, in the following we present the implementation of the classes whose instantiation makes up a software sporadic terminal object.

The requirement that software sporadic terminal objects provide one single method named \texttt{Start} fully determines the specification of the corresponding generic unit, which is shown in figure 5.

OBOSS active components communicate by message passing, as a reflection of which the parameter of the \texttt{Start} operation is expected to be a packet-encoded message with a variable inner structure. The parameter type determines the type of the incoming message, that is, the inner structure of the corresponding packet. The sporadic operation of the task (Operation) takes a parameter of the same type as the \texttt{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 (and consequently makes it software sporadic). Figure 6 shows the implementation of the sporadic task class. It is worth noting that the OBOSS implementation allows for the occurrence of exceptions and performs some standard handling on them. Owing, in part, to limitations with the language support in use and to design decisions, the current OBOSS implementation ties the exception handling policy at this level to the specific OBOSS build. Future OBOSS upgrades may look into whether and how such a policy could be rendered more easily configurable.

The Start operation, which HRT-HOOD wants asynchronous and non-suspending, 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 software sporadic task will have its own event buffer with a given size (Event_Buffer_Size) and a given ceiling priority (Event_Buffer_Priority).

Figure 7 shows the implementation of the class Queue, from which the protected terminal object Event_Buffer is derived. This implementation determines the type of data-oriented synchronisation model in place between the caller of the Start operation and the sporadic task itself.

In the current implementation, the Start operation is non-suspending even on full buffer while the Extract operation suspends the caller indefinitely on empty buffer.

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

All building blocks of this kind sit at the bottom of the OBOSS reuse hierarchy and correspond to auxiliary objects in the taxonomy introduced earlier. As such, they represent instances of 100% unmodified reuse. All the configuration parameters (‘attributes’, ‘terminals’ and ‘connectors’ as shown in figure 3) they need to operate with would descend down to this level, and to all other intermediate levels concerned, from a user specification provided at the outermost level of the OBOSS system.
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 => Parameter_Type,
 Queue_Size => Event_Buffer_Size,
 Queue_Priority => Event_Buffer_Priority);

function Start (Parameter : Parameter_Type) return Boolean is
begin
  return Event_Buffer.Deposit (Parameter);
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 : Parameter_Type;
  Activation_Time : Ada.Real_Time.Time := <...> -- configurable by user-level parameters
begin
  delay until Activation_Time; -- to ensure orderly task activation after elaboration
  loop
    Due_Event := Event_Buffer.Extract;
    Operation (Due_Event);
  end loop;
exception
  when ... -- appropriate handling
end Sporadic_Task;
end Sporadic_Task;

Figure 6. OBOSS implementation of sporadic task class (body).

Application Process Breakdown View

The notion of application process is absolutely central to our interpretation of the PUS model. The OBOSS design of the application process confers it the modularity, the loose-coupling and the composeability that make PUS systems inherently scalable.

An OBOSS application process is built up by composition and configuration of reuse modules according to a recurrent design pattern. We illustrate this constructive approach focusing on a selected sample of reuse modules. Figure 8 displays the details of the breakdown that we shall present to this end.

At the top of the application process hierarchy we find a very simple interface specification, which we show in Figure 9.

Our implementation defines the Send_Packet operation exported by the object in terms of an operation with the same profile exported by a child object. This process of encapsulation reflects the HOOD hierarchical decomposition approach.
Figure 7. OBOSS implementation of queue class.

Procedure Send_Packet represents the sole entry point for incoming service requests destined to the application process. In terms of our earlier taxonomy of OBOSS parameters, this procedure represents a ‘connector’.

Child object AP_i.PacketDispatcher implements this connector. In this implementation we encounter the first instance of embedded concurrency within OBOSS. The body of the package in figure 10 shows how OBOSS makes objects active by attaching the desired concurrent behaviour to their definition. Also note how OBOSS uses the user package AP_i.Parameters to hold all the actual parameters (‘attributes’) required to configure the various constituents of Application_Process_i.

A call to Application_Process_i.Send_Packet resolves into a call to the Start operation of the sporadic task TC_Forwarder embedded in AP_i.PacketDispatcher. We have seen earlier that the implementation of the Start operation entails a transaction on the event buffer embedded within the
Figure 8. OBOSS architecture of an application process: breakdown of data flow and control structure for command execution and reporting.
with PUS; -- OBOSS reuse component
package Application_Process_i is
  procedure Send_Packet (Packet : in PUS.PUS_Packet); -- for message producers to invoke
end Application_Process_i;

with AP_i_Packet_Dispatcher -- to handle incoming messages
package body Application_Process_i is
  procedure Send_Packet (Packet : in PUS.PUS_Packet) is
    begin
      AP_i_Packet_Dispatcher.Send_Packet (Packet); -- to start the processing of the incoming message
      Send_Packet;
    end
end Application_Process_i;

Figure 9. OBOSS implementation of application process template.

with PUS; -- OBOSS reuse component
package AP_i_Packet_Dispatcher is
  procedure Send_Packet (Packet : in PUS.PUS_Packet);
end AP_i_Packet_Dispatcher;

-- a number of OBOSS reuse components including:
with AP_i_TC_Interpreter;
with AP_i_Parameters; -- holds all configuration parameters for the application process
package body AP_i_Packet_Dispatcher is
  package TC_Fowarder is new Sporadic_Task
    (Task_Priority => AP_i_Parameters.Task_Priority,
     Task_Stack_Size => AP_i_Parameters.Task_Stack_Size,
     Event_Buffer_Priority => AP_i_Parameters.Event_Buffer_Priority,
     Event_Buffer_Size => AP_i_Parameters.Event_Buffer_Size,
     Operation => AP_i_TC_Interpreter.Receive_Packet,
     Parameter_Type => PUS.PUS_Packet, ... -- all other parameters);
  procedure Send_Packet (Packet : in PUS.PUS_Packet) is
    begin -- non-blocking
      if not TC_Fowarder.Start (Packet) then
        ... -- error handling on full buffer
      end if;
      Send_Packet;
    end AP_i_Packet_Dispatcher;

Figure 10. OBOSS implementation of packet dispatcher template.
with PUS; -- OBOSS reuse component
package AP\_i\_TC\_Interpreter is
    procedure Receive\_Packet (Packet : in PUS\_PUS\_Packet);
end AP\_i\_TC\_Interpreter;

-- various OBOSS reuse components including for example:
with AP\_i\_Service\_J; -- instantiation of PUS service J for application process i
package body AP\_i\_TC\_Interpreter is
    procedure Reject\_Packet (Packet : in PUS\_PUS\_Packet) is -- to handle erroneous messages
end Reject\_Packet;
    procedure Receive\_Packet (Packet : in PUS\_PUS\_Packet) is
begin
    case Get\_Service\_Type (Packet) is
    -- OBOSS reuse service function
    -- for as many PUS services as the application process supports, e.g.:
        when Service\_J => AP\_i\_Service\_J.Receive\_Packet (Packet);
        when others => Reject\_Packet (Packet);
    end case;
end Receive\_Packet;
end AP\_i\_TC\_Interpreter;

Figure 11. OBOSS implementation of packet interpreter template.

sporadic task instance. This choice allows us to fully decouple the message sender from the message receiver, so that the two may effectively proceed in parallel.

The implementation of procedure Send\_Packet lets the sender proceed even in case of full buffer. 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 define as generic actual parameters such notions as: the operation that the embedded thread must carry out in response to the triggering event; the priority at which that has to be done; the ceiling priority and, for software sporadic objects only, the size of the associated event buffer. The instantiation of the software sporadic task TC\_Forwarder typifies the OBOSS approach in this respect.

Package AP\_i\_TC\_Interpreter defines the Receive\_Packet operation of TC\_Forwarder. This operation multiplexes service requests to the appropriate service provider within the application process. This reflects the delegation of control approach taken by OBOSS, which allows service providers to be implemented as active objects and thus to operate concurrently.

All the units we have seen down to this level correspond to template objects in the taxonomy introduced at the top of the section. OBOSS provides models of such units, which the user needs to adapt to the specific characteristics, topology and nomenclature of the system. Only a small proportion of this model code will produce instances of 100\% unmodified reuse.

One further level down in the OBOSS hierarchy we find the implementation of the PUS service capabilities embedded within the application process. This is where OBOSS pushes its reuse approach
to its extreme. As we see from figure 8, \texttt{AP\_\_Service\_J} is described as an active (software sporadic) object. In HRT-HOOD parlance this means that the object may embed multiple threads the activity of which triggers on a software synchronisation event.

Figure 8 shows that the implementation of operation \texttt{Receive\_Packet} exported by unit \texttt{AP\_\_Service\_J} involves the concurrent operation of at least one software sporadic task. Several others would also appear at a finer level of decomposition. 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 \texttt{Service}\_\_J shown in figure 12 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. Unit \texttt{AP\_\_Service\_J} still corresponds to a template object, whereas unit \texttt{Service}\_\_J represents a base class, which yields 100\% unmodified reuse.

‘Terminal’ parameters in OBOSS take the form of \texttt{Send} or \texttt{Receive} subprograms. Specific reuse components invoke these subprograms within the execution of the service protocol concerned. OBOSS however does \textit{not} employ the invocation of these subprograms as the triggering event of any of its built-in tasks. OBOSS users therefore have to supply non-suspending implementations of these ‘terminal’ subprograms in order for the calling tasks to retain conformance with the Ravenscar Profile, which only allows a single activation event per task.

Service components use ‘terminal’ parameters of this kind to acquire values from remote units or to send control commands to them. For reasons of conciseness, we omit showing instances of ‘terminal’ parameters in the code fragments that follow.

The instantiation of the formal generic operation \texttt{Deposit} of the \texttt{Service}\_\_J class shown in figure 12 is the mandatory ‘connector’ parameter that dispatches the messages produced by the operation of the service object to the Packet Router.

Like all PUS services, \texttt{Service}\_\_J features a high degree of internal concurrency. As figure 8 shows, the HRT-HOOD connotation of this concurrency is software sporadic. The activity of this object is in fact instigated by the arrival of a service request, which occurs by invocation of the \texttt{Receive\_Packet} procedure. The implementation of this base class encompasses a considerable amount of recurrent active and passive components.

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. The implementation of operation \texttt{Receive\_Packet} exported by unit \texttt{Service}\_\_J resolves in depositing the service request into the event buffer of \texttt{Command\_Interpreter.Interpreter}. In turn, the arrival of the service request triggers the activation of the corresponding software sporadic task and the execution of the \texttt{Execute\_Command} procedure, whose declaration we see in figure 12. In this way, the service execution may proceed in parallel to any other unrelated activities in the system.

Individual application processes in OBOSS may embed multiple message producers. Those messages typically contain service reports, which are produced on execution of the \texttt{Execute\_Command} procedure.

All packet-based service reports produced by an application process have to be posted into the mailbox of the Packet Router, which is the sole entity with knowledge of all legal destinations in the system. To this end, a dedicated passive object per application process takes care of the relevant packet collection and dispatching. The base object in charge of this operation (an auxiliary object for our taxonomy) is named \texttt{Packet\_Depositor}. Figure 13 shows its implementation.
Various OBOSS reuse components, including examples:

```plaintext
package AP_iPacket_Depositor is
  procedure Deposit(Packet : in PUS.PUS_Packet);  -- mandatory connector
end AP_iPacket_Depositor;
```

```plaintext
package Service_J is
  procedure Receive_Packet(Packet : in PUS.PUS_Packet);
end Service_J;
```

```plaintext
-- several instantiations of recurrent support components, including examples:
package Command_Parser is new Command_Parser(<<configuration parameters>>);
package Command_Interpreter is
  procedure Receive_Packet(Packet : in PUS.PUS_Packet);
end Command_Interpreter;
```

```
package body Service_J is
  generic
    with function Deposit(Packet : in PUS.PUS_Packet) return Boolean;
  end Service_J;
```

```
package body Command_Interpreter is
  procedure Receive_Packet(Packet : in PUS.PUS_Packet);
begin
  the caller does not block!
  if not Interpreter.Start(Packet)
    . . . -- error handling on full buffer
  end if;
end Receive_Packet;
```

```
package body Command_Interpreter is
  procedure Execute_Command(Packet : in PUS.PUS_Packet);
  package Interpreter is new Sporadic_Task(.Operation => Execute_Command, . . .);  -- error handling on full buffer
  procedure Receive_Packet(Packet : in PUS.PUS_Packet) is
    if not Interpreter.Start(Packet) then
      . . . -- error handling on full buffer
    end if;
end Receive_Packet;
end Command_Interpreter;
end Service_J;
```

Figure 12. OBOSS implementation of service class and instantiation template.

Obvious reasons of conciseness prevent us from going any further in the illustration of OBOSS implementation. The fraction that we have shown in this section has the least domain-specific characteristics. This selection illustrates a design approach that could arguably apply to any producer-consumer architecture. A large proportion of the OBOSS code that we have not shown here is far more domain-specific. In fact, OBOSS achieves most of its unmodified reuse potential at domain-specific level. It is thanks to the design pattern that we have illustrated above, however, that those domain-specific reuse-intensive modules can fit a scalable and composeable system architecture.
with ... -- a number of OBOSS reuse components including:
with Packet_Depositor;
with Packet_Router; -- the system-level message router
package AP Packet_Depositor is new Packet_Depositor
   (Deposit_Response => Packet_Router.Deposit);
with PUS;
generic
   with function Deposit_Response (Packet : PUS.PUS_Packet) return Boolean;
package Packet_Depositor is
   function Deposit (Packet : PUS.PUS_Packet) return Boolean;
end Packet_Depositor;

with ... -- a number of OBOSS reuse components
package body Packet_Depositor is
   ... -- several service operations
   function Deposit (Packet : PUS.PUS_Packet) return Boolean is -- administration on outgoing Packet
      -- which the actual generic parameter attaches to the Deposit operation of the Packet Router
      return Deposit_Response (Packet);
end Packet_Depositor;

Figure 13. OBOSS implementation of packet depositor class and instantiation template.

OBOSS Flavours of Object Orientation

In our discussion of the OBOSS implementation strategy we have frequently implied the existence of an OBOSS object model of some sort. We now want to better characterise this notion of object with respect to its present and planned extent of object orientation.

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. With Reference [20], we single out 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. In fact, compliance with the Ravenscar Profile only allows server-side synchronisation, for it prescribes inter-task communication to solely occur via the mediation of protected objects, which effectively operate as the server object for both sides of communication.

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. Reference [20] 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 to wait for up to a maximum time for the constraint to be met, returning with an error otherwise. Whereas the Ada language definition [23] 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 full buffer or empty buffer.

The use of HRT-HOOD led OBOSS to model message receivers as sporadic objects. As the design method prescribes, the exported operation Extract of the base class Queue shown in figure 7 maps to a protected entry with blocking response on the state constraint violation of empty buffer. For the state constraint violation of full buffer on sending, instead, OBOSS has opted for a balking response for compliance 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 full buffer and typically results in the issue of an exception message directed toward a supervisory authority. This behaviour is in accord with what is generally expected of current on-board systems. Earlier in this section we have encountered two code fragments that give rise to balking response behaviour: (i) in the implementation of operation Send_Packet of object AP_i_Packet_Dispatcher (cf. figure 10); (ii) in the implementation of operation Receive_Packet of object Command_Interpreter within class object Service_J (cf. figure 12).

Table I summarises the data-oriented synchronisation characteristics of OBOSS objects. We maintain that the synchronisation behaviour of OBOSS objects fully meets the applicable requirements of the target application domain.

As we mentioned earlier, the present generation of OBOSS classes can be instantiated but cannot be extended. This limitation bounds the reusability potential of OBOSS and weakens the object orientation characteristics of its design.

The Queue class presents an obvious case where specialisation would be more powerful and advantageous than instantiation. Its current design suffers from two intrinsic limitations (cf. [6]):

- 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;
Table I. Data-oriented synchronization characteristics of OBOSS.

<table>
<thead>
<tr>
<th>synchronization constraints</th>
<th>enforcement side</th>
<th>enforcement style at sender</th>
<th>enforcement style at receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>exclusion</td>
<td>server</td>
<td>total mutual exclusion</td>
<td>total mutual exclusion</td>
</tr>
<tr>
<td>state</td>
<td>server</td>
<td>balking</td>
<td>blocking</td>
</tr>
<tr>
<td>coordination</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>

- 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 the insertion and extraction policy for use by selected software sporadic objects with the respective event buffers. The current implementation fixes the choice to a single solution (e.g.: FIFO, priority based) for all objects within any OBOSS build. This is no intolerable drawback at present, but it may become more of an annoyance in the face of increasing user demand. The issue whether OBOSS includes other components as obviously and as conveniently amenable to extension as the event buffer is the subject of current investigation. The sole downside to unabridged adoption of inheritance by extension lies in the increase in resource requirements that could incur from it, particularly with respect to memory size and predictability of execution.

An obvious opportunity for OBOSS to benefit from the expressive power of Ada 95 object orientation stems from the use of tagged types in the place of discriminated records for the handling of polymorphic data objects.

The case for tagged types 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. Compilers use the heap when they cannot statically determine the size of the return parameters of subprograms. The use of the heap however constitutes a potential hazard to the stability of non-terminating programs. Tagged types whose interpretation depends on the context of execution (which Ada calls `run-time dispatching`) may incur the use of the heap. Any increase in the heap-intensiveness of the treatment of tagged types would obviously undermine their usefulness to OBOSS.

As a general principle and in spite of the intellectual interest of unabridged object orientation, OBOSS revisions will always give precedence to the preservation of the current integrity, predictability and resource demand of the system.

---

1 Tagged types are the closest Ada 95 gets to the primitive notion of ‘class’ of other object-oriented languages. The inheritance hierarchy rooted at Ada 95 tagged types matches the derivation class of those languages. This view is orthogonal to the notion of OBOSS class used in this paper, which is rooted at the generic package, the ultimate building block of Ada programs.
The challenge for a software reuse framework in the application domain of OBOSS is to respond to seemingly contrasting needs.

The chief reuse objective was one of generality, to maximise the unmodified reuse potential while allowing users to adopt the service profile that fits their needs. The major acceptance criterion for on-board software systems in the class of OBOSS, however, focuses on predictability of execution and containment of memory requirements.

To achieve the reuse objective, OBOSS has to cover a large cross-section of the PUS service catalog and of the variety of its configuration parameters. To qualify for acceptance, the OBOSS computational model must warrant predictability in the face of scalability and the OBOSS implementation technology must make moderate demands for memory.

In the following we analyse how our implementation responded to these needs. We first look at how compliance with the Ravenscar Profile impacts the scalability and the predictability of the system. We then analyse the tribute the OBOSS generality pays to the memory budget of the system. Finally, having determined the acceptability of the technical criteria, we devise a projection technique to predict the ratio of unmodified reuse attainable by OBOSS users.

The Ravenscar Profile as a Limiting Factor

The decision to conform the OBOSS concurrency model to the prescriptions of the Ravenscar Profile had so vast repercussion on its design and implementation that it obviously needs assessment. Earlier in this paper we have briefly recalled the restrictions emanate from the definition of the profile.

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 traditionally followed by a large fraction of the on-board software community. Ironically, they are even more permissive with regard to the use of native Ada tasking (which was often substituted by custom forms of concurrency).

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 shows that the profile suits well the needs of on-board real-time embedded systems of the new generation. Moreover, our earlier discussion of the OBOSS implementation also shows that the profile provides the object world of OBOSS with an elegant concurrency model.

Task Density

The combined effect from the OBOSS reuse philosophy and the Ravenscar tasking model calls for a radical ‘specialisation’ 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 it 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 old Ada technology (i.e., Ada 83) usually contemplated. This phenomenon was fully apparent in our implementation.
A large task population poses system designers a serious threat in two, arguably orthogonal, dimensions: predictability of execution and scalability of architecture.

OBOSS solves both problems efficiently and elegantly thanks to the combined effect of two key features of its design. It earns scalability from the HRT-HOOD computational model, the active terminal objects of which are threaded in accord with the Ravenscar Profile and consequently are freely composeable. It earns predictability from the power of the theories of schedulability associated with the Ravenscar Profile (cf. e.g.: [3]) and from the regularity of its tasking architecture that feeds them the necessary information.

OBOSS users can determine exact bounds to the tasking population of the system for any variation of its configuration parameters. In fact, those bounds can be expressed in a fairly elegant fashion, as we illustrate in the following.

Table II lists the structural components of OBOSS that feature internal concurrency. Equation 1 shows how those components contribute to the determination of the tasking population in an OBOSS-based system comprised of local application processes and agents for remote application processes. $I_m$ in the equation is an $[m \times 1]$ identity matrix, which we use to reduce the result to vectorial form.

$$TP(m, a) = OI + AI + SF \times AS \times I_m$$

Each structural component of the equation is expressed in term of a $(\tau, s, c, w)$ tuple, which respectively denotes the number of interrupt sporadic threads, software sporadic threads, cyclic threads and protected objects to account for the component.

With equation 1 we can easily determine the lower bound and the upper bound to the tasking population for all variants of OBOSS systems.

The lower bound is obtained on a system comprised of one agent associated to the TC and TM end of the system (whereby $a = 1$) and one single application process embedding no PUS services (whereby $m = 1$). One such application process — and the resulting system — alone would serve no useful purpose other than acting as a TC destination and a TM producer. For instance, this would be good enough for carrying out electrical integration of the system.

The upper bound is obtained when every application process in the system embeds the whole set of PUS services supported by OBOSS, each holding one protected state vector.

Eqn. 1 sets the lower-bound value to $TP_{\text{min}} = [1, 5, 0, 12]$; whereas for the upper-bound value we obtain: $TP_{\text{max}}(m, a) = [a, (1 + 2a + 9m), 4m, (6 + 4a + 17m)]$. 

---

**Table II. Notations.**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$TP$</td>
<td>total tasking population</td>
</tr>
<tr>
<td>$OI$</td>
<td>system infrastructure component</td>
</tr>
<tr>
<td>$AI$</td>
<td>application process infrastructure component</td>
</tr>
<tr>
<td>$SF$</td>
<td>service framework descriptor</td>
</tr>
<tr>
<td>$AS$</td>
<td>application process service descriptor</td>
</tr>
</tbody>
</table>
These bounds confirm the predominantly event-driven nature of control flow in OBOSS and, arguably, in all on-board systems faced with PUS requirements. We note that all the tasks in term $TP_{\text{min}}$ classify as sporadic, while only $4m$ periodic tasks contribute to term $TP_{\text{max}}(m, a)$.

We reckon that an average OBOSS system would comprise in the lower region of 40 tasks and well over 40 protected objects, thus needing some 80 or more 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 (e.g.: by way of avoidance of any form of round-robin scheduling for equal-priority tasks) and, consequently, to that of the scheduling analysis.

The Ada Reference Manual [23] is not prescriptive on the issue of the supported priority range and leaves the decision to the language implementor. Current Ada technology for 32-bit embedded systems typically support 256 priorities (which would be more than adequate for OBOSS) except when it interfaces to foreign operating system technology, where it is the latter to determine the priority range, which it often limits to 32.

OBOSS embeds concurrency directly within its reuse model. This reflects the HRT-HOOD connotation of it, which adds tasking characteristics to the functional properties of its objects. This approach raises the computational power of the reuse components and permits users to determine the tasking population of an OBOSS instantiation for any variation of its configuration parameters.

**Sizing Requirements**

An important dimension of our evaluation has to be the impact of the OBOSS reuse model on the sizing budget of the system. On-board memory is a cost driver for satellite systems in at least two distinct respects: the material cost of its radiation-hardened technology and the demand it places on the power budget of the system. On-board software applications consequently face stringent limitations on the amount of memory required to host and run the system. Any new software technology for use with on-board software applications will always be challenged for its impact on memory and so will OBOSS.

Two aspects are especially relevant to our evaluation of this particular issue: (a) the size inflation caused by the inception of the PUS view of DHC systems, the generality and flexibility of which are intrinsically more demanding than traditional tailored-to-needs systems; (b) the size inflation caused by the OBOSS reuse approach with respect to functionally equivalent systems with less emphasis on reuse.

Consideration of aspect (a) tells us what cost we have to expect, in the particular respect of on-board memory, upon the transition to the PUS. This cost shall show inferior to the promise of development and operational benefits for the new approach to be of industrial interest. To estimate this cost we relate the code size of a traditional (pre-PUS) DHC system with minimum functionality with the code size of an OBOSS system with the corresponding PUS functionality. In PUS terms, this corresponds to a single application process supporting a classic set of 3 optional services. In establishing the comparison we need to discount the differences in baseline technology used by the respective systems. The traditional system targeted a 16-bit processor with a CISC-like instruction set. The corresponding OBOSS system targets a 32-bit RISC processor. We discount this difference by using a normalisation factor of 3 to denote that a program written for the former architecture will become (about) 3 times bigger when compiled for the latter. We infer this factor on consideration of the doubling of the word size and the more verbose programming style incurred on the transition to 32-bit pipelined architectures.
Table III. Impact of PUS Model and OBOSS Reuse Approach on Code Size (in bytes)

<table>
<thead>
<tr>
<th>component</th>
<th>pre-PUS 16-bit Ada</th>
<th>OBOSS PUS 32-bit Ada</th>
<th>other PUS 32-bit C</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime support</td>
<td>4,308</td>
<td>14,580</td>
<td>11,420</td>
</tr>
<tr>
<td>platform services</td>
<td>21,470</td>
<td>25,506</td>
<td>11,380</td>
</tr>
<tr>
<td>system infrastructure</td>
<td>6,492</td>
<td>90,780</td>
<td>12,624</td>
</tr>
<tr>
<td>application process</td>
<td>18,788</td>
<td>79,876</td>
<td>14,760</td>
</tr>
<tr>
<td><strong>total</strong></td>
<td><strong>51,058</strong></td>
<td><strong>210,742</strong></td>
<td><strong>50,184</strong></td>
</tr>
<tr>
<td><strong>PUS inflation factor</strong></td>
<td><strong>1.00</strong></td>
<td><strong>1.38</strong></td>
<td><strong>0.33</strong></td>
</tr>
<tr>
<td><strong>OBOSS reuse inflation factor</strong></td>
<td>—</td>
<td><strong>4.20</strong></td>
<td><strong>1.00</strong></td>
</tr>
</tbody>
</table>

Consideration of aspect (b) gives an indication of the size overhead that results from the OBOSS emphasis on reuse and its selection of enabling technology. This size overhead shall have to be modest for OBOSS to attract users and deflect them from opting for a custom implementation of the system. To address this dimension we compared the same OBOSS system considered for aspect (a) with a non-OBOSS PUS system developed with non-Ada technology for the same target family but with a much lower reuse objective.

Table III shows the details of our evaluation, which we broke down in accordance with the main architectural components identified in figure 2. We attach sufficient trust to the breakdown figures displayed for non-OBOSS systems because we could easily relate their functional constituents to the OBOSS nomenclature. We find a breakdown at this level is useful to identify the system components that exhibit the most variability in memory requirements. As we said earlier, we apply the normalisation factor to the pre-PUS values shown in the table.

The Ada-to-Ada comparison shows an inflation factor of 1.4, which we may ascribe to the PUS overhead. In particular, the break-down figures in table III show how apparent the PUS overhead becomes for the system infrastructure component that is most implicated with the interpretation of the PUS communication protocols.

The Ada-to-C comparison shows an inflation factor of 4.2, which reflects the considerable overhead of the OBOSS reuse model and the greater language complexity of Ada. This overhead originates in the application-specific components of the system. The Ravenscar runtime used for OBOSS almost equals in size the stripped runtime used for pre-PUS Ada system and pays a little overhead tribute to the real-time kernel used for the C system. OBOSS moves specialisation to the low-level tier of the system (the platform services component) which can be tailored to needs and therefore restrain its size overhead. Conversely, OBOSS moves reuse-gared flexibility to the higher-level tiers of the system, which consequently pay a significant overhead tribute to custom implementations.

The PUS-related inflation factor denounces a wholly acceptable overhead, which the many comparative advantages of the PUS service model richly pay for.
The OBOSS-related inflation factor challenges the OBOSS reuse model. The overhead factor for which the OBOSS implementation is directly responsible can only be responded by a major increase of reuse value with respect to less reuse-gearied competitor technology.

We will see in the following section that OBOSS features a potential for unmodified reuse that arguably pays off its tribute to size overhead.

**Reuse Ratio**

The promise of unmodified reuse needs quantification. Project managers need quantitative information to assess the pros and cons of design baselines, especially when they involve the adoption of software reuse frameworks with the characteristics of OBOSS.

Next to the design and implementation of OBOSS we have developed a method for the determination of the unmodified reuse ratio that users can attain. In the following we illustrate the rationale and the product of the method.

From a reuse perspective, an OBOSS-based system comprises two basic categories of constituents: (i) components that form part of the OBOSS reuse infrastructure and the user simply needs to configure for the system in question, typically by way of generic instantiation; (ii) components that are platform-specific or system-specific, and that the user needs to provide, typically in the form of actual generic parameters or as bodies of prescribed specifications or else as self-standing components with no direct interface to OBOSS reuse assets. For the purpose of this discussion we may regard an OBOSS-based system as a cascading hierarchy of such components.

At the top of the hierarchy we find the OBOSS notion of PUS system, which encompasses:
(a) the OBOSS reuse infrastructure at system level, which allows the system to hang together;
(b) user-supplied services that enable the system to perform platform-specific operations, like writing to and reading from physical devices;
(c) a battery of \( n \) application processes \( (n \geq 1) \) which embed the set of PUS service capabilities specifically required of the system.

One level down in the hierarchy, the OBOSS notion of application process \( i \) \( (1 \leq i \leq n) \) includes:
(d) the OBOSS reuse infrastructure at this level, which allows the application process to hang together and includes a set \( M_i \) of service capabilities implemented as generic instantiations;
(e) a collection of user-supplied modules that define the system-specific attributes of the application process, including any functional capabilities not prescribed by the PUS standard.

At the bottom level of the hierarchy, a generic service instantiation \( i_j \) \( (j \in M_i) \) of application process \( i \) comprises:
(f) the OBOSS reuse infrastructure at service level;
(g) the user-supplied definition of the generic actual parameters required for the service.

The service capability set \( M_i \) of application process \( i \) is of cardinality \( m_i \) \( (m_{\text{min}} \leq m_i \leq m_{\text{max}}) \) where \( m_{\text{min}} \) denotes the minimum number of services required by the standard per application process and \( m_{\text{max}} \) denotes the total number of services allowed by the standard. Individual services are uniquely identified by an index in the range \( [1, m_{\text{max}}] \) so that \( M_i \) is effectively an ordered set of indices in that range. The current version of the PUS standard prescribes that \( m_{\text{min}} = 2 \leq m_i \leq m_{\text{max}} \leq 256, \forall i \).

Our method determines the value of \( s \) (with \( 0 \leq s \leq 1 \)), that is, the level of unmodified reuse potentially attained in the development of an OBOSS system build, relative to the total size of the system.
Table IV. Components of an OBOSS-based system.

<table>
<thead>
<tr>
<th>level</th>
<th>component</th>
<th>origin</th>
<th>size</th>
<th>instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>infrastructure</td>
<td>OBOSS</td>
<td>$\sigma_{Ir}$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>platform services</td>
<td>new</td>
<td>$\sigma_{Ir}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>system infrastructure</td>
<td>OBOSS</td>
<td>$\sigma_{Ir}$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>new</td>
<td>$\sigma_{Ir}$</td>
<td></td>
</tr>
<tr>
<td>application process i</td>
<td>infrastructure add-on</td>
<td>OBOSS</td>
<td>$\sigma_{iRi}$</td>
<td>$1 \leq i \leq n$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>new</td>
<td>$\sigma_{iRi}$</td>
<td></td>
</tr>
<tr>
<td>service $ij$ ($j \in M_i$)</td>
<td>infrastructure add-on</td>
<td>OBOSS</td>
<td>$\sigma_{iRij}$</td>
<td>$m_{\min} \leq m_i \leq m_{\max}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>new</td>
<td>$\sigma_{iRij}$</td>
<td></td>
</tr>
</tbody>
</table>

The structure of an OBOSS-based system is inherently such that $g$ increases with the density of OBOSS-supported PUS services per application process and with the number of such application processes in the system.

Table IV itemises the components of an OBOSS-based system indicating the hierarchical level at which they place, the respective supplier, the symbol denoting their size and their recurrence at that level.

All of the OBOSS-specific symbols in table IV that denote size have statically known values, which represent permanent attributes of the relevant reuse assets. For example, the system infrastructure component ($\sigma_{Ir}$) presently amounts to about 5,000 source statements. For the non-OBOSS components, one would do with size projections for new components, or with actual data for existing ones. In general, the non-OBOSS component providing the so-called platform services (cf. figure 2) represents the largest unitary component of the system, with a size ($\sigma_{P\epsilon}$) that may approach the lower region of 10,000 source statements for systems of the new generation.

Using the actual values associated to the size symbols of table IV, which we omit for the sake of simplicity, we determine the overall estimated size of an OBOSS-based system (which we denote $g$) and, from that, the amount of unmodified OBOSS reuse achieved within it (which we denote $\sigma$).

For every OBOSS reuse component, the size of the unit contributes to the running total of unmodified reuse. Similarly, for every user-supplied component, the size of the unit contributes to the running total of the required development. The ‘instances’ indicator of table IV determines how many times the contribution of individual components needs to be accounted for at that hierarchical level.

We have studied the behaviour of $g$ under the hypothesis of a mean density of PUS services per application process within a system populated with multiple application processes. We have chosen a moderate $m_i$ value for $i$ in the range $[1, n = 10]$ and we have assumed the usual OBOSS coverage of those services, in the region of 80% per application process. (That is, for an application supporting 5 services, 4 of them usually happen to match the OBOSS offer.)

\n
5Note that for traditional systems $\sigma_{P\epsilon}$ hardly exceeds 5,000 source statements on account of the considerably less stringent demand for autonomous capabilities of on-board fault tolerance.
Figure 14 plots the resulting $s$ over an x-axis that denotes $\sigma$, which increases with the number of application processes in the system.

The plot in figure 14 exhibits a logarithmic behaviour with $s$ approaching 79% for system sizes in excess of 100,000 source statements (equivalent to a very large system with over $n > 10$ application processes). For more typical values of $\sigma$ in the region of 25,000 source statements and $n_{\text{leq}}$, $s$ approaches 60%. We contend that this level of unmodified reuse is a target worthy of industrial interest and a sufficient bonus for the size overhead paid to less reuse-geared approaches. The benefit of the OBOSS reuse model especially shows as we note that OBOSS promises this level of unmodified reuse to a wide range of system configurations, which is well outside the reach of custom implementations.

CONCLUSION

In this paper we have presented an engineering approach to software reuse that attained a considerable potential for unmodified reuse in conjunction with a high degree of architectural scalability.

Our engineering approach looked at software reuse from the three complementary and inter-related perspectives of ‘reuse by architecture’, ‘reuse by artifacts’ and ‘reuse by technology’.

The emergence of a domain-specific standard especially tailored to our application domain greatly facilitated our endeavour, particularly with regard to the architecture and artifacts dimensions.
The embedding of Ravenscar concurrency within Ada generics was the technology solution that earned our software architecture an excellent blend of scalability and predictability. The combined use of both language features proved a powerful and expressive means of implementation.

As a reflection of a code legacy largely rooted in the object based connotation of Ada 83, our software is not a champion of pure object orientation. This limitation did not preclude us the ability to factorise a large proportion of our code efficiently. Yet, analysis is on-going at the time this paper is being written to determine what aspects of language-level object orientation could be advantageous to our concept.

In order to ease the transposition of the engineering approach presented in this paper to application domains other than ours we deliberately restrained the discussion of domain-specific aspects.

**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**