

# A Case Study in the Reuse of On-board Embedded Real-Time Software

Tullio Vardanega<sup>1</sup>, Gert Caspersen<sup>2</sup>, and Jan Storbak Pedersen<sup>2</sup>

<sup>1</sup> European Space Agency Research & Technology Centre,  
Keplerlaan 1, 2200 AG Noordwijk, Netherlands  
tullio@ws.estec.esa.nl

<sup>2</sup> TERMA Elektronik AS,  
Bregnerodvej 144, 3460 Birkerød, Denmark  
{gpc, jnp}@terma.com

**Abstract.** The rise of the ‘cheaper, faster, better’ mission paradigm increasingly challenges the industrial development of satellite systems. The novel paradigm will have a profound impact on the production of the real-time software embedded on board new-generation systems. This paper discusses how software reuse may fit in the rising development scenario and how reuse interacts with other important players in the picture, especially the software process model and the on-board software architecture.

## 1 The Rise of a New Project Paradigm

Similarly to a variety of other technology segments, the industrial development of satellite systems is confronted with the rise of the ‘cheaper, faster, better’ mission paradigm. The novel paradigm denotes the urge to attain increased operation capability, decreased power consumption, lower launch mass and shorter time to market. This trend will cause the future development scenario of on-board embedded real-time systems to be dominated by the demand to: (i) reduce the development schedule from the present 3-4 years to 18-24 months; (ii) deliver better mission product via increased autonomy and responsiveness of operation; (iii) support increasingly more software-intensive systems.

These demands have a vast impact on the architecture of the spacecraft bus, i.e. the module in charge of the control services on board the satellite. For example, the software size of those components is predicted to rise from the present 8-12,000 source statements to the 15-25,000 of new missions under the same language baseline. In the face of this event, the productivity of the software development process shall have to at least quadruple, for twice as big software will have to be delivered in half the time. Concurrently, the operation requirements placed on the software product will increase the functional complexity and the real-time criticality of the system, which will have to be attained within no less than the present envelope of dependability.

Earlier work (cf. [1, 2]) has argued that an iterative and incremental development process is better equipped than the traditional ‘waterfall’ model to cope with the increasing complexity of the product and the concurrent reduction of the development schedule. This paper concurs with this vision and takes it further towards industrial practice.

The envisioned process calls for a software architecture flexible enough to accommodate successive increments but also capable of assuring the integrity of the system and the convergence of the process. In our view, however, three further ingredients are required for a highly-productive realisation of the process: (i) an application model that favours the scalability of the architecture; (ii) a software reuse framework that accelerates the construction of systems in match with that application model; and (iii) enabling technology that supports the production of reusable, reliable and predictable software, with Ada [3, 4] playing a crucial rôle in this respect.

We recently reviewed the performance of a pilot development centred around those three ingredients in the frame of a project funded by the European Space Agency. This paper presents some of our findings.

## 2 Process Model

As outlined in [2], the process model assumed in this paper bases on two conceptual pillars: the design framework and the computational model.

The design framework covers the segment of the development process that spans from the software requirements phase to the detail design phase. Contrary to the classical waterfall model, the design framework contemplates the possibility of multiple, incremental iterations across phases. The notion of design framework aims to master the anticipated occurrence of feedback-based iterations in the design and verification of new-generation systems. The incremental nature of the process is the means to break the overall development down to (parallel) threads of sub-development, each characterised by bounded complexity and greater likelihood of faster completion.

The computational model assembles the notions that steer the iterative process within the design framework and lead it to convergence. Those notions defines: (a) the type of components to be used in the construction of the real-time architecture of the system, the means for the communication and synchronisation between them, and the concurrency paradigm assumed for their execution; (b) the execution properties and real-time attributes that characterise those components as well as the constraints placed on their use and interaction; and (c) an underpinning analytical model that allows the static analysis of the real-time behaviour of systems constructed in terms of those components. These definitions collectively determine the abstract view of the architectural support for predictability availed to the process. Suitable computational models support the construction of flexible yet statically verifiable architectures that fit in the incremental nature of the development and reduce the extent of real-time verification needed at integration testing.

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 [5, 6] and bases on the principles of fixed priority preemptive scheduling [7, 8]. HRT-HOOD extends the base HOOD design method [9] in use at the European Space Agency by incorporat-

ing the abstractions supported by the revised tasking model of Ada 95 in the fashion prescribed by the ‘Ravenscar profile’ [10].

Our computational model contemplates: periodic objects that model time-triggered threaded activities; sporadic objects that model event-triggered threaded activities; and protected objects that model non-threaded structures for mutually exclusive access to shared data. 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. [2] describes a set of static analysis techniques that support this notion.

### 3 Application Domain

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 bus. This set of communication-related services is collectively termed Data Handling Control (DHC) system.

DHC services are typically executed in response to requests disjointly arriving from either the ground centre, for the processing of incoming TC, or on board, for the issue of TM or the servicing of other requests. Accordingly, DHC services have a predominantly sporadic, event-driven nature. Well-defined operation requirements normally bound the inter-arrival time of the triggering events for any given mission scenario. Additionally, DHC systems may also include a small number of naturally periodic activities. For example, the monitoring of selected on-board parameters with respect to mission-specific surveillance criteria, and the execution of any required corrective actions.

For traditional systems, the ground centre needs to maintain at all times the most accurate information about the status and operation of the spacecraft. New-generation autonomous systems tend to move part of the control responsibility to the space segment. Either way, the control activity assumes the minimisation in the latency and jitter incurred between the issue of a command and its actual execution on board. Most of the real-time requirements on DHC services should thus be regarded as mission-critical. Failure to meet any of these requirements may decrease the mission product or even compromise the mission.

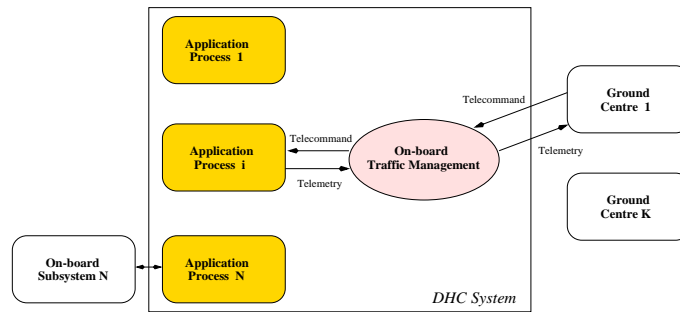
## 4 Project Ingredients

### 4.1 Application Model

The European Space Agency’s Packet Telemetry and Packet Telecommand Standards [11, 12] and the CCSDS Recommendations from which they are derived [13, 14] address the end-to-end transport of TC and TM data between user applications on ground

and application processes on board. The European Space Agency’s Packet Utilisation Standard (PUS) [15] complements and extends those standards by defining a packet-based interface for the execution of certain services by on-board application processes. The PUS defines those services by prescribing the service model (i.e. how the application processes are to behave on arrival of a given service request) and the inner structures of the associated TC and TM packets. In the absence of those definitions, mission teams traditionally opted to develop their own operational concept, along with a set of mission-specific TC and TM format layouts. That approach effectively prevented any practical reuse of DHC software as well as of electrical ground support equipment and control centre infrastructure. The PUS specifically emerged as an attempt to control the earlier ad-hoc approach to satellite operations.

In the context of the PUS, an application process is defined as an on-board entity capable of receiving TC packets and generating TM source packets. An application process is uniquely and statically identified for the entirety 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. We retain this basic definition and assume that multiple application processes may reside on the same processing node. The PUS operational model is shown in Figure 1.



**Fig. 1.** Packet Utilisation Standard Operational Model.

The original definition of the PUS addressed the requirements arising from the commissioning and operation phase of the satellite life cycle as seen from the perspective of the ground centre(s). Recent work [16] has shown the usefulness of applying the PUS service model to the communications between on-board application processes without any necessary involvement of the ground. An on-going revision of the PUS standard is presently addressing the implications of this extension [17]. New space projects looking into autonomous operation of the spacecraft (e.g.: [18]) are also considering exploiting this potential.

Important advantages may be gained from the consistent application of the PUS service model throughout all phases of system definition and implementation. As shown in Fig. 1, a DHC architecture structured according to the partitioning implied by the PUS subdivides into a set of loosely-coupled application processes. This subdivision

naturally reverberates across the development philosophy itself. Specification, implementation and verification of a DHC system thus conceived may in fact break down to progressive increments of sub-development, each addressing individual application processes or their next level of integration. This notion fits exceptionally well in the process model proposed in Ref.s [1] and [2].

## 4.2 Software Reuse Framework

Several key features of a PUS-compliant DHC software architecture emanate directly from the operational model shown in Fig. 1. First and foremost, the bulk of the software infrastructure in charge of on-board communications that adhere to the standard packet definitions may stay the same across missions. Moreover, the loosely-coupled character of the PUS operational model and the event-driven nature of the associated service model imply that the processing of independent commands may proceed in parallel throughout the system as a whole but also within individual application processes.

Consequently, the software architecture of application processes themselves naturally break down in two major components: a recurrent infrastructure for the support of the internal routing of commands and responses; and an application-specific implementation of the service capabilities to be provided by that application process.

These notions embody a vast amount of reuse potential. The on-board operations support software (OBOSS) system [16] was developed as a software reuse framework aimed to exploit that potential in the construction of PUS-based DHC software architectures.

OBOSS provides easily configurable support for packet-based message passing between the ground segment and on-board application processes. The DHC architecture promoted by OBOSS is loosely-coupled and naturally 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. The communications architecture within OBOSS is a star configuration, based on mailboxes and packet-encoded messages, and centred around a Packet Router. The Packet Router determines the destination of the incoming message and deposits the packet in the mailbox of the relevant application process. OBOSS provides each application process with a local Packet Dispatcher that fetches the packet from the private mailbox and delivers it to the service capability concerned. OBOSS also allows application processes to operate as communications agents for on-board subsystems physically disjoint from the DHC. In that case, the DHC-resident application process embeds no service capability and simply forwards packets to and from the associated remote subsystem. Packet-encoded messages carry either commands for the execution of certain services within a given application process; or data produced from the execution thereof. A Ground Interface component transforms the source TC incoming from ground into internal packets carrying the corresponding command and passes them on to the Packet Router. Similarly, the Ground Interface receives from the Packet Router internal packets carrying responses directed to ground and transforms them into source TM. Fig. 2 depicts the main architectural components of a DHC software system based on OBOSS.

On top of the reference architecture depicted in Fig. 2 and the corresponding communications infrastructure, OBOSS provides a set of generic service capabilities that

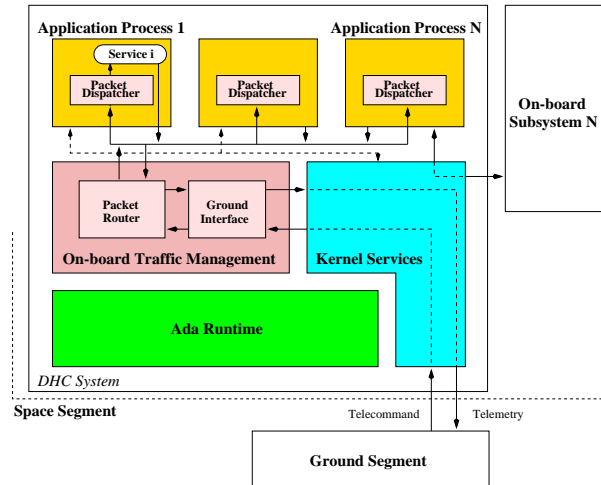


Fig. 2. Software Architecture Breakdown of DHC System based on OBOSS.

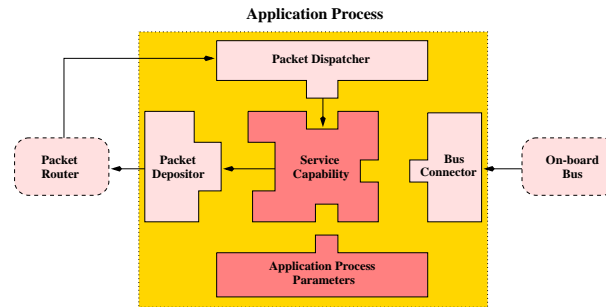
cover a subset of those specified in the PUS, and a vast amount of guidance information for the construction of mission-specific DHC systems.

Parameter configuration is the primary means offered by OBOSS to assemble reusable software artifacts into a coherent system architecture. The OBOSS parameters belong in three main categories: ‘terminals’, which interface individual application components to the OBOSS communications infrastructure; ‘attributes’, which capture specific functional characteristics of the individual component; and ‘connectors’, which encapsulate dependencies on the physical architecture of the DHC system under construction. The box tagged Kernel Services in Fig. 2 globally represents all the needed ‘connectors’.

Fig. 3 depicts a simplified break-down of the software components that collectively constitute an application process embedding the incarnation of an OBOSS generic service capability. With respect to the cited parameter categories, Fig. 3 shows: a Packet Dispatcher (‘terminal’), which delivers to the service incarnation incoming messages that carry commands for that service; a Packet Depositor (‘terminal’), which forwards responses or commands generated by the service incarnation to the Packet Router; specific parameters (‘attributes’), which define the characteristics of the service to be provided by the incarnation embedded in that particular application process: e.g.: unique identifier of the application process, service policy, service priority; a Bus Connector (‘connector’), which interfaces the application process and the service incarnation to specific remote units on board the spacecraft.

### 4.3 Enabling Technology

Ada 83 [3] has been the programming language of choice for the vast majority of European Space Agency’s space projects to date. On the availability of mature support for embedded targets, Ada 95 [4] will be the natural choice for future projects.



**Fig. 3.** Incarnation of OBOSS Service Capabilities in an Application Process.

The choice of Ada as enabling technology fits perfectly in the highly-productive process model sought by our experiment in several distinct respects. The OBOSS architecture embodies a vast amount of internal event-driven concurrency that needs to be controlled for efficiency, responsiveness and predictability. The loosely-coupled character of the OBOSS message-passing infrastructure needs to rely upon efficient asynchronous communications. The Ada implementation of the Ravenscar profile provides excellent support for both demands. Even though rarely used in on-board systems, Ada generics provide the most direct means to implement the OBOSS generic service capabilities. Accordingly, the desired characteristics of the service incarnation are simply specified as the actual parameters of the generic units concerned. One dimension of our experiment was expressly to assess the practicality of extensive use of Ada generics in resource-constrained on-board embedded real-time systems.

The strong industrial connotation of our undertaking dictated the use of Ada 83 technology equipped with support for the Ravenscar profile. As new technology becomes available, we also want to assess the benefit of transitioning the OBOSS implementation to Ada 95 in view of its greater expressive power and support for extensibility.

## 5 Evaluation

We recently conducted an experiment to evaluate the performance of a software development process centred around the ingredients described in section 4.

The idea behind the experiment was to engage a third-party team in the implementation of a sizeable proportion of a new-generation DHC system compliant with the PUS and based on the OBOSS software reuse framework. The third-party team had no prior knowledge about the PUS and no prior experience with OBOSS.

We selected several performance indicators and measured their evolution across the experiment. In the following we discuss some of the observations made during the experiment.

### 5.1 Productivity

The first dimension we looked into was the productivity rate achieved in the experiment with respect to typical values observed in industrial projects. The experiment

encompassed the development, integration and verification of one application process embedding a sizeable proportion of the OBOSS-supported PUS services. We measured the rate in terms of ‘new’ source Ada statements delivered per hour by the team across a comparable segment of the software development process (i.e. from design to verification). Our definition deliberately excluded from the measurement the amount of unchanged bespoke software that was part of the experimental system. That precaution was necessary as the size of the OBOSS framework exceeded 70% of the total size of the experimental system, and the observed duration of the experiment was insufficient for the team to master the framework in its entirety, so that we could not regard it as truly ‘delivered’.

Even with those precautions, we observed an important increase in productivity. We attributed the increase to several concurrent factors; in particular: (i) the lesser load imposed on the team by our incremental process; and (ii) the reduced design and integration effort incurred on adoption of the OBOSS architectural framework.

Fig. 4 relates the productivity attained in the experiment to the productivity band and width of observation for traditional (precursor) developments as well as recent commercial projects.

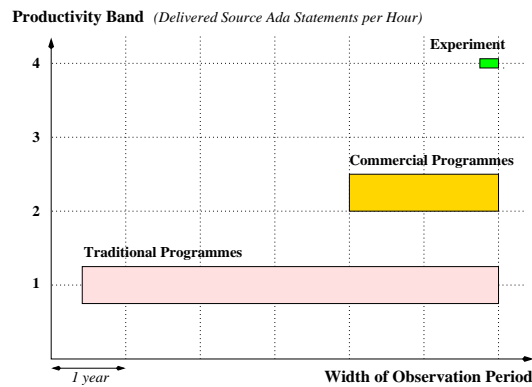


Fig. 4. Software Productivity Rate in Experiment vs Typical Space Projects.

## 5.2 Memory Size Requirements

The second dimension of our observation was the impact of the OBOSS reuse framework on the memory size requirements of the system. A certain inflation in code size is the inevitable ‘dark’ side of any ambitious reuse initiative. It certainly was in our experiment. In Table 1 we relate the code size of the experimental system based on OBOSS to a functionally comparable traditional DHC implementation. The notion of ‘functionally comparable’ is rather weak in this context, for no traditional DHC to our knowledge gets anywhere near the processing requirements induced by the PUS. Yet, we picked one last-generation traditional DHC as the reference term.

**Table 1.** Impact of OBOSS on Memory Budget: Code Size (in bytes)

<b>component</b>	<b>traditional DHC 16-bit CISC</b>	<b>OBOSS-based DHC 32-bit RISC</b>
<i>Ada runtime</i>	4,308	73,404
<i>kernel services</i>	21,470	53,313
<i>communications services</i>	6,492	76,112
<i>application layer</i>	18,788	79,028
<b>total</b>	51,058	281,856
<b>inflation rate</b> (normalised)	1.00	1.84

For the traditional DHC we considered the code breakdown on a 16-bit CISC space processor. For the OBOSS-based DHC we considered the experimental implementation on a new-generation RISC space processor. We then normalised the observed size increase to the ‘natural’ inflation rate incurred on the transition from 16-bit CISC to 32-bit RISC, which normally places in the region of 3.0. The breakdown components listed in Table 1 correspond to the main architectural blocks identified in Figure 2. The ‘application layer’ component denotes the equivalent of one application process embedding a sizeable proportion of the OBOSS-supported PUS services.

The experimental system was fully coded in Ada and executed on a standard runtime for the lack a ‘Ravenscar-only’ runtime suited for the target processor. The traditional system included a mixture of Ada and assembly and run on a custom (stripped down) version of the standard Ada runtime.

### 5.3 CPU Load Requirements

The subsequent dimension we looked into was the impact the OBOSS reuse framework had on the CPU load budget of the system. The circumstances described above made it difficult to establish a truly equivalent load scenario to relate against. Furthermore, the ‘traditional’ DHC operated on the basis of a fixed cyclic schedule, whereas OBOSS operates on preemptive scheduling. On account of this event, the nominal application scenario under consideration had: a ‘worst-case’ load, given by the fixed width of the DHC slot; and an ‘actual’ load, given by the effective utilisation of the DHC slot incurred in the scenario. We contrasted the figures thus determined to those respectively obtained from worst-case static analysis of the OBOSS-based system and from actual execution of the scenario.

Table 2 relates the values obtained in the two cases, after normalisation of the CPU power ratio between the two processors, which we estimated to a factor of 6 in favour of the 32-bit RISC.

### 5.4 Task Density

The decision to adopt the Ravenscar profile in the OBOSS implementation naturally resulted in raising the task density of the system to unprecedented levels. This was no

**Table 2.** Impact of OBOSS on Nominal CPU Load Budget (%).

CPU load	traditional DHC 16-bit CISC	OBOSS-based DHC 32-bit RISC	inflation rate (normalised)
worst-case	26.70	8.23	1.85
actual	7.01	7.54	6.45

great deal, for the expressive power and semantic contents of our computational model ensure full control of the induced concurrency. Yet, we found it crucial to demonstrate that well-defined bounds exist to the tasking population produced by an OBOSS-based system. In fact, those bounds can be expressed in a fairly elegant fashion, as follows.

**Table 3.** Definitions

$TP$	total tasking population
$CF$	communications framework component
$AC$	application process interface component
$SF$	service framework descriptor
$AS$	application process service descriptor

Table 3 lists the structural components of OBOSS that feature internal concurrency. Eqn. 1 defines how those components contribute to the determination of the tasking population in an OBOSS-based system comprised of  $m$  application processes, denoted AP.

$$TP(m) = CF + AC + SF \times AS \times I_m \quad (1)$$

$$\text{where: } SF = \begin{bmatrix} \tau_1 & \tau_2 & \tau_3 & \tau_4 & \tau_5 & \tau_6 & \tau_7 \\ \pi_1 & \pi_2 & \pi_3 & \pi_4 & \pi_5 & \pi_6 & \pi_7 \end{bmatrix} = \begin{bmatrix} 3 & 2 & 2 & 1 & 1 & 1 & 0 \\ 4 & 2 & 3 & 1 & 1 & 1 & 1 \end{bmatrix} \quad (2)$$

with  $\tau_i$  and  $\pi_i$  respectively denoting the number of tasks and protected objects required to implement PUS service  $i$  for any given OBOSS implementation (currently in the range [1..7]);

$$CF = \begin{bmatrix} 3 \\ 7 \end{bmatrix} \quad \text{and} \quad AC = m \times \begin{bmatrix} 1 \\ 2 \end{bmatrix} \quad \text{using the same notation;} \quad (3)$$

$$AS = \begin{bmatrix} e_{1,1} & \dots & e_{1,m} \\ \dots & & \\ e_{6,1} & \dots & e_{6,m} \\ n_1 & \dots & n_m \end{bmatrix} \quad \text{with } e_{i,j} = \begin{cases} 1 & \text{if AP}_j \text{ embeds service } i \\ 0 & \text{if AP}_j \text{ does not embed service } i; \end{cases} \quad (4)$$

$n_j$  is the number of packet stores used by AP <sub>$j$</sub> ; and  $I_m$  is an  $[m \times 1]$  identity matrix.

With Eqn. 1 we easily determine the lower bound and the upper bound to the tasking population for all variants of OBOSS-based DHC systems that comply with the PUS.

The lower bound is obtained on a system comprised of one single application process (whereby  $m = 1$ ) embedding no PUS services (whereby  $\mathbf{AS} = [0_{1,1}, \dots, 0_{6,1}, 0_1]$ ), hence merely acting as destination of source TC and producer of source TM. The upper bound is obtained when every application process in the system embeds the whole set of PUS services supported by OBOSS (whereby  $\mathbf{AS} = [1_{1,1}, 1_1, \dots, 1_{6,m}, 1_m]$ ). Eqn. 1 sets the lower-bound value to:  $\mathbf{TP}_{min} = [4, 9] + [1, 1]$ , where the latter term accounts for the TC interpreter in the application process; and the upper-bound value to:  $\mathbf{TP}_{max}(m) = [(3 + m \times 11), (5 + m \times 15)]$ . As an indicator of the predominant event-driven control flow in OBOSS, we note that all the tasks in term  $\mathbf{TP}_{min}$  are classified as sporadic, whereas only  $m \times 4$  periodic tasks contribute to term  $\mathbf{TP}_{max}(m)$ .

## 6 Lessons Learned

The level of software productivity required by new-generation space projects compels industry to abandon the one-off product concept for the notion of ‘product in a series’. Software reuse may play a crucial rôle in this transition.

OBOSS was conceived, designed and developed as a means to faster and progressively more reliable production of new-generation DHC software systems. The experimental observations discussed in section 5 indicate that this objective may be achieved, even in the case of inter-company reuse. The experiment showed that the software productivity, expressed in terms of new source Ada statements delivered per hour, may rise in excess of 4 times the traditional level and 2 times the performance of recent commercial projects (cf. Fig. 4). This productivity boost comes at a cost, though. The experiment highlighted some visible cost elements: the increase by a factor of 2 in the size of memory (cf. Table 1) as well as in the worst-case CPU load provisions (cf. Table 2). These indicators are important for on-board embedded systems that are subject to stringent resource constraints due, in part, to the material cost and, in part, to the proportional rise of the verification cost. In our view, the experimental results obtained in this respect yield modest concern. The impact on memory size needs to be assessed in the context of specific project economics: flight memory is normally a costly and spared resource; yet, the productivity increase and other architectural considerations may factor the extra cost off. The impact on CPU load is probably only of statistical interest in the face of the rising availability of more powerful space-qualified processors.

There are other, less visible costs that also need to be computed in the equation.

On-board embedded systems are exposed to levels of verification that require of the supplier the technical mastership of all components of the system, irrespective of their origin. This requirement may prove an unsurmountable hurdle as the implications it carries are not thoroughly understood at the start of the project. As well-structured as it may be, OBOSS brings along an unprecedented amount of flight-worthy bespoke software.

OBOSS represents up to 70% of a software-intensive new-generation DHC system. Thus, it takes a certain amount time to acquire a sufficient mastership of its internals and operation. The iterative and incremental nature of the proposed process model surely aids in spreading the ‘learning curve’ over an affordable time span. Yet, the effort needed to do so must be budgeted as an investment on a foreign product. This provi-

sion is likely to take a level of commitment that goes beyond the scope of the project itself. The major challenge facing OBOSS is, thus, to demonstrate its actual strategic convenience also in the case of inter-company reuse.

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

## References

1. Vardanega, T., van Katwijk, J.: Productive Engineering of Predictable Embedded Real-Time Systems: The Road to Maturity. *Information and Software Technology*, **40** (1998) 745–764.
2. Vardanega, T., van Katwijk, J.: A Software Process for the Construction of Predictable On-Board Embedded Real-Time Systems. *Software - Practice and Experience*, **29**:3 (1999) 1–32.
3. ISO, Ada Reference Manual. International Standardisation Organisation ISO/IEC JTC 1/SC22, Geneva, Switzerland (1987). ISO/IEC 8652:1987.
4. ISO, Ada Reference Manual. International Standardisation Organisation ISO/IEC JTC 1/SC22, Geneva, Switzerland (1995). ISO/IEC 8652:1995.
5. Burns, A., Wellings, A.: HRT-HOOD: A Structured Design Method for Hard Real-Time Systems. *Real-Time Systems*, **6** (1994) 73–114.
6. Burns, A., Wellings, A.: HRT-HOOD: A Structured Design Method for Hard Real-Time Systems. Elsevier Science, Amsterdam, Netherlands (1995).
7. Audsley, N., Burns, A., Richardson, M., Wellings, A.: Hard Real-Time Scheduling: The Deadline Monotonic Approach. *Proc. Real-Time Operating Systems and Software*, IEEE, **8** (1991) 127–132.
8. Audsley, N., Burns, A., Wellings, A.: Deadline Monotonic Scheduling Theory and Application. *Control Engineering Practice*, **1**:1 (1993) 71–78.
9. HTG, HOOD Reference Manual 3.1. HOOD Technical Group, Prentice Hall (1993).
10. Baker, T., Vardanega, T.: Session Summary: Tasking Profiles. *Ada Letters*, **XVII**:5 (1997) 5–7. *Proc. 8<sup>th</sup> Int'l Real-Time Ada Workshop*.
11. ESA, Packet Telemetry Standard. European Space Agency, Noordwijk, Netherlands, PSS-04-106: Issue 1 (1988). (<http://esapub.esrin.esa.it/pss/pss-cat1.htm>)
12. ESA, Packet Telecommand Standard. European Space Agency, Noordwijk, Netherlands, PSS-04-107: Issue 2 (1992). (<http://esapub.esrin.esa.it/pss/pss-cat1.htm>)
13. CCSDS, Telemetry Summary of Concept and Rationale. Consultative Committee for Space Data Systems, CCSDS 100.0-G-1: Issue 1 (1987). (<http://www.ccsds.org/publications.html#telemetry>)
14. CCSDS, Telecommand Summary of Concept and Service. Consultative Committee for Space Data Systems, CCSDS 200.0-G-6: Issue 6 (1987). (<http://www.ccsds.org/publications.html#telecommand>)
15. ESA, Packet Utilisation Standard. European Space Agency, Noordwijk, Netherlands, ESA PSS-07-101 Issue 1 (1994). (<http://esapub.esrin.esa.it/pss/pss-cat1.htm>)
16. CRI: Onboard Operations Support Software - Modules Users Manual. Deliverable on ESTEC Contract 11277/94/NL/FM(SC), European Space Agency, Noordwijk, Netherlands (1997). (<http://ftp.estec.esa.nl/pub/ws/wsd/oboss/www/oboss.html>)
17. Parkes, A., Kaufeler, P., Merri, M., Valera, S., Vardanega, T.: The Future of the Packet Utilisation Standard. *Proc. 1<sup>st</sup> ESA Workshop on Tracking, Telemetry and Command Systems*, European Space Agency (1998).
18. Teston, F., Creasey, R., Van der Ha, J.: PROBA: ESA's Autonomy and Technology Demonstration Mission. *Proc. Int'l Astronautical Congress, International Aeronautical Federation* **48** (1997).