



PROJECT

**Software System Development
for SC Data Handling & Control**

TITLE

**Report of WP 5300:
Third Party Evaluation of Approach for Reuse**

Ref: ESTEC Contract 12797/98/NL/PA

	<u>Name</u>	<u>Function</u>	<u>Date</u>	<u>Signature</u>
Prepared :	Björn Enoksson Håkan Svanberg	Software Engineer Software Engineer		
Checked :	Mikael Thorvaldsson	Software Manager		
Authorised :	Stefan Fredriksson	Project Manager		
Distribution Complete :	DFE, DP, DP-BE, DP-HH			
Summary :				

Reg. Office:
Saab Ericsson Space AB
S-405 15 Göteborg
Sweden
Reg. No: 556134-2204

Telephone:
+46 31 735 00 00
Telefax:
+46 31 735 40 00

Linköping Office:
Saab Ericsson Space AB
S-581 88 Linköping
Sweden

Telephone:
+46 13 18 64 00
Telefax:
+46 13 13 16 28



Class : Deliverable
Contract No :

Host System : Microsoft Word 97 for Windows, SE Macro Rev 3.0
Host File : ...\\D-D-REP-0006_01-SE.doc

SUMMARY

This report is the result of a third party evaluation of approach for reusing the Onboard Operations Support Software (OBOSS).

DOCUMENT CHANGE RECORD

Changes between the current issue and the previous one are marked with a left-bar, except for changes that are purely editorial.

Issue	Date	Paragraphs affected	Change information	Reference
1	22 Nov 1999	All	New document	



TABLE OF CONTENTS		PAGE
1.	INTRODUCTION	4
1.1	Purpose	4
1.2	Document Overview	4
1.3	References	5
1.4	Glossary	5
1.5	Abbreviations.....	5
2.	SOME GENERAL ASPECTS ON REUSE	6
3.	ON REUSING OBOSS.....	8
3.1	Requirements Analysis and Definition.....	9
3.2	Software Design	9
3.3	Coding	10
3.4	Testing	11
3.5	Maintenance.....	12
3.6	Management	13
3.7	Quality Assurance and Control.....	13
4.	AN IMAGINARY USE CASE BASED ON SOHO COBS	14
4.1	Introduction	14
4.2	COBS Overview	14
4.2.1	Context	14
4.2.2	Tasks.....	16
4.2.3	Architecture and Scheduling	17
4.2.4	Implementation.....	17
4.2.5	Development Model.....	17
4.3	The Imaginary COBS Project.....	17
4.3.1	TC/TM handling.....	18
4.3.2	PUS Applicability	19
4.3.3	Compiler.....	20
4.3.4	Target	20
4.4	Estimated Software Sizes	20
4.4.1	Model	20
4.4.2	Kept and Replaced Parts of COBS.....	22
4.4.3	Adding PUS Functionality	22
4.4.4	Size of OBOSS and Related Adaptations	23
4.4.5	Resulting Sizes	24
4.4.6	On the Validity of the Result.....	24
4.5	Saving in the Project.....	24
4.6	Recurrent PUS Projects	25
5.	CONCLUSION.....	26



1. INTRODUCTION

This report is the result of a study to perform a so-called third party evaluation of an approach for reusing the Onboard Operations Support Software (OBOSS); OBOSS 2 is here applicable to this study. In the sequel, OBOSS 2 is called shortly OBOSS unless otherwise is said.

This study is performed within the project

Software System Development for SC Data Handling & Control

This project is managed by TERMA, Denmark.

Saab Ericsson Space (SES) is responsible for the following work package:

WP 5300: Third Party Evaluation of Approach for Reuse

Ref: ESTEC Contract 12797/98/NL/PA.

1.1 Purpose

The purpose of the first part of this study is to address the economic implications of reusing the OBOSS (architecture and components) compared with an in-house development of a data handling system (from scratch) based on the Packet Utilisation Standard (PUS). This standard is defined in [PUS].

In the second part of this study, software project is investigated by evaluating the effect OBOSS would have had on the software development in a data handling system (COBS) already developed by SES.

This study is made with the approach that SES has the role of being a presumptive user of OBOSS.

1.2 Document Overview

This report comprises the following main parts:

- Chapter 1 is this introductory chapter containing purpose, references, and used abbreviations.
- Chapter 2 discusses some general aspects on reusing software.
- Chapter 3 contains a comparison of two scenarios: one in which OBOSS is reused and another in which own-development is chosen.
- Chapter 4 describes an imaginary use case that is based on the software COBS in the SOHO data handling system.
- Chapter 5 finalises the report with a conclusion of this study.



1.3 References

- [CaseStudy] A Case Study in the Reuse of On-board Embedded Real Time Software
Tullio Vardanega (ESTEC), Gert Caspersen (TERMA), and Jan Storbank
Pedersen (TERMA)
- [PSS-05-0] ESA Software Engineering Standards
ESA PSS-05-0, Issue 2, February 1991
- [PUS] Packet Utilisation Standard
ESA PSS-07-101, Issue 1, May 1994

1.4 Glossary

- Customer stands for a customer in a presumptive project that comprises a spacecraft data handling system based on PUS.
- Developer is synonymous with user of OBOSS.
- Supplier means the developer and supplier of OBOSS.
- User means the user of OBOSS. In this study, SES is assumed as a user of OBOSS.

1.5 Abbreviations

- COBS Central On-Board Software
- OBOSS Onboard Operations Support Software
- PUS Packet Utilisation Standard (PUS)
- SC Spacecraft
SES Saab Ericsson Space
SOHO Solar and Heliospheric Observatory
- TC Telecommand
TM Telemetry



2. SOME GENERAL ASPECTS ON REUSE

In this section, some general aspects are discussed concerning the reuse of software developed by an external supplier.

The main reason for reusing an already developed software product is the economic profit that might be obtained. The supplier's technical and other experiences give, of course, confidence in the software product but if no (or insignificant) cost savings are obtained, the reuse is not a real alternative from the economical point of view.

However, there might be exceptions from the economic profit as the main factor. In safety critical systems, for instance, the confidence in software products that are already in use, can be decisive.

When a presumptive user shall make a decision to reuse a software product, an economic assessment and risk analysis must be done. The cost analysis together with the risk analysis for the reuse approach, are compared with the cost and risks of the approach that the user makes the software product.

The following items might be candidates for risks (the list is not exhaustive):

- The interfaces do not fit user's application; e.g. the hardware interface does not fit or the application process model does not fit with the application of the user.
- Memory constraints
- Performance requirements
- Potential misunderstandings in the description or usage

Taken into account these items, a plan should be established in order to estimate for each identified risk, the amount of effort that is required to mitigate or eliminate the risk.

In order to consider reuse of a software product it must normally fulfil the following requirements:

- The software must be well known for the user and easy to access.
- The software must be well defined and fit the user's requirements: functional, performance, and quality.
- The software must be properly tested and documented.
- The responsibility must be clear concerning the software maintenance.
- There must be an economic gain from the software reuse.

The work that is required for the reuse approach has to be compared with the work that is needed to develop the software in-house. An estimation has to be performed of the size of the software and what activities or phases in the development cycle that are not needed when reusing a foreign software.

It must be emphasised that a "new" development in-house seldom starts from scratch. Normally, a certain degree of reuse is made in such an approach: experiences from other similar projects are utilised even if no units (modules) can be reused "as is".



A reused software component can contain so-called unused code (i.e. code that shall not be executed/used). For certain applications, e.g. safety/critical software, this must be taken into account. Unused code is normally not allowed in safety/critical applications. It is, accordingly, important to investigate whether generic software meets the applicable quality requirements.

3. ON REUSING OBOSS

The main task of this study is to evaluate which cost savings might be obtained when in a project using OBOSS instead of developing a new data handling system. Thus, the following two scenarios are compared:

- **Scenario A – “Reuse”**
OBOSS is reused; possible modifications and additions are made.
- **Scenario B – “New”**
All software is developed in-house; no part of OBOSS is reused.

The following prerequisites for the two scenarios are assumed:

- PUS puts the requirements on the spacecraft data handling system.
- The customer provides a specification of the product to be developed.
- The software is coded in Ada.
- The development shall comply with the ESA Software Engineering Standards, [PSS-05-0] (or ECSS-E40).

Further, it is also presumed that SES is a presumptive developer of a software product (data handling system) in both scenarios.

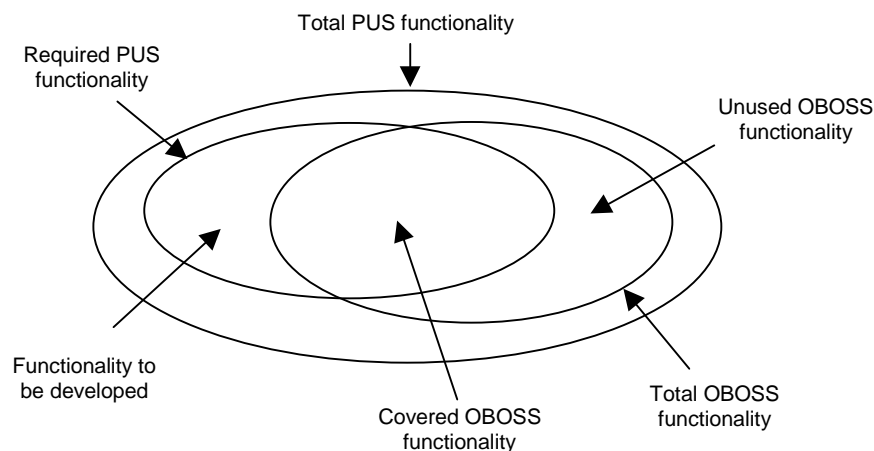


Figure 1 - Functionality coverage

Given a certain application, this application requires the functionality “Required PUS functionality” as shown in Figure 1. This functionality is in scenario A more or less covered by OBOSS; called “Covered OBOSS functionality” in the figure. Thus, the functionality not covered by OBOSS must be developed. It is here assumed in-house by the user. In addition, OBOSS itself might be modified. Whether all “Unused OBOSS functionality” can fully be removed might be dependent on the situation (this has not been analysed in this study).

In scenario B, the “Required PUS functionality” must be developed in-house (own-development).

The following main activities (or processes) of a normal software life cycle are discussed below in order to find candidates for cost savings:



1. Requirements analysis and definition
2. Software design
3. Coding
4. Testing
5. Maintenance

The activities of management, quality assurance, and quality control are also treated below.

3.1 Requirements Analysis and Definition

Scenario A

The developer/user has to analyse the customer's need and has to define the specific requirements on the software (e.g. requirements on the implementation). An analysis of OBOSS must be done to check whether it covers all needed functionality and has appropriate performance. If it does not, complements and/or changes must be defined.

The access to the provided OBOSS documentation and the specifications might lead to a better understanding of the requirements (and the customer's need), but an analysis and definition of the requirements must still be made. Thus, the areas "Functionality to be developed" and "Covered OBOSS functionality" in Figure 1 must be determined.

Scenario B

Normal work is done to analyse the problem and to define the requirements on the software to be developed.

Costs

The number of activities to be performed in this process Requirements Analysis and Definition is probably of the same for both scenarios in average projects. However, the amount of work per activity should be dependent on the specification provided by the customer.

The cost might be lower for A than for B if the required PUS capabilities are close to what is supported by OBOSS; i.e. few modifications and/or complements must be done. Costs should be saved if the customer specifies clearly which requirements belong to "Covered OBOSS functionality" and which do not.

If OBOSS supports a smaller part of the required PUS capabilities, i.e. the area to "Covered OBOSS functionality" is relatively small, the cost of scenario A will be of the same magnitude as the cost of scenario B.

3.2 Software Design

Scenario A

An adequate documentation of OBOSS is here assumed to be available. This documentation should comprise as an absolute minimum: the user's manual (including information on the architecture of OBOSS) and a description how to adapt OBOSS.



The OBOSS architecture is here used. If additional units must be designed, it is integrated with the new-designed software.

The detailed design is roughly limited to those units of OBOSS that must possibly be modified and/or to those possible units with a functionality that is not provided by OBOSS.

General OBOSS support packages may be utilised in the rest of the design.

In safety/critical applications, "unused code" has to be taken into consideration. Investigations might result in that some parts of the reused software must be modified.

Scenario B

Normal software design of all units to be developed must be done. Of course, experiences from other projects with similar projects are "reused" (design patterns, units, etc). The design will be reviewed in work-through meetings.

Costs

For a reused unit, no design efforts are needed, of course. The cost that might be saved should be approximately proportional to the amount of reused software units.

The benefit of reuse might be reduced due to the following:

- If OBOSS is very different from the company design patterns, the reuse potential for the rest of the software may be limited (both import and export), or major structure-related adaptations are required.
- The presumed (by OBOSS) software platform (Ada tasking, "passive" tasks, generics) may not be available or feasible, which might require broad modifications to the active and protected OBOSS objects. However, in the modular architecture of OBOSS, the means of execution are well isolated, limiting the work associated with transition to other execution environments.
- Required function and real time properties of platform adaptations may not fit well with existent drivers, protocols or hardware, making adaptations more complex.

3.3 Coding

Scenario A

The amount of coding work in this scenario is dependent on the number of OBOSS units that must be modified and on the units that must be developed (to complement OBOSS).

Scenario B

Normal coding is performed for all required units. Each unit will then be inspected (i.e. in so-called code inspections).

Costs

The amount of coding work is dependent on the number of units that must be modified and/or new developed. The cost savings might be approximately proportional to the number of reused software units (or lines of reused code). In addition to the coding itself of the reused units, especially code inspections can be eliminated, assumed the development approach/model is the same.



The benefit of reuse might be reduced due to the following:

- The extensive use of generics tends to cause resource problems with either memory or execution time, depending on the selected method for instantiation.
- The large number of tasks required may be a problem. It may be necessary to reduce the number of tasks.

3.4 Testing

Testing comprises here three test steps (or processes): unit (or module) testing, integration testing, and system testing (sometimes also called validation or qualification testing by various users).

Scenario A

The developer must do unit testing for any modified OBOSS units as well as for new developed units. Reused (non-modified) generic OBOSS units must also be unit-tested to verify that the instantiation of generic OBOSS units works properly.

Non-generic OBOSS units are expected to be unit-tested by the OBOSS supplier. The supplier should also test generic units in comprehensive instantiations.

The extensive use of tasks and generics may complicate unit testing, depending on the applicable criteria on test method.

Full integration testing must be performed.

Support for this testing could be provided through test stimuli/results definition files (packet level data, driver response data) which then could be transformed by the developers for the actual test environment.

Full system testing must be performed.

The OBOSS supplier should perform the system testing of OBOSS, using a full-fledged example of instantiation.

Scenario B

All three tests must be executed in accordance with the requirements (standards).

Costs

For scenario A, unit testing is a candidate for cost reduction. Depending on the level to which the supplier has tested a unit, this test activity might be relaxed or even removed. This must be clearly expressed in the beginning of the project.

The integration testing should be easier to execute in scenario A, especially in case when there are relatively few modified and/or new developed units, presumed that the reused software is clearly documented. Thus, the cost savings should be related to the relative numbers of reused units and modified/new units.



There are probably less significant cost savings identified for system testing for A compared with B. How much the cost can be reduced should be dependent on the level of the support that the OBOSS supplier can provide in form of test suites, etc.

The possible cost savings are dependent on which tests the customer requests and on which test software the supplier of OBOSS provides.

3.5 Maintenance

Scenario A

The supplier of OBOSS has to define in a maintenance plan (or similar) which tasks of preventive maintenance as well as corrective maintenance the supplier intends to perform. The maintenance must be defined very early in the project, preferably before start.

If it is expected that the user of OBOSS shall perform some updates (corrections) of the reused software the user must have access to the associated test software.

Generally, technical support from the supplier as well as future availability to source code and build tools must be considered because of often very long maintenance requirements, perhaps 10-20 years. As OBOSS is delivered as source code, however, no such problems should exist.

Scenario B

The developer has to perform the preventive and corrective maintenance that is applicable according to the contract.

Costs

For scenario A, it is very important that those maintenance tasks the supplier of OBOSS respectively the user of OBOSS shall perform, are clearly defined in the beginning of a project. If some of the maintenance tasks are not defined in an unambiguous way or have not been identified when the project starts, situations might arise resulting in that the cost for this activity will become higher than estimated and exceed possible economical margins.

For the OBOSS part of the product, the corrective maintenance is expected to be low, compared with the modified/new part.

For scenario B, the developer/user has better control of the maintenance.

It is difficult to find any significant cost savings when running the project according to scenario A. The cost for maintenance is probably lower for B than for A when taken into account that the supplier must support the OBOSS part of the software product.



3.6 Management

Scenario A

The same development process model will be followed in this scenario as the one applied to scenario B; i.e. the same number of activities will be performed. For instance, the number of reviews is not reduced in a “reuse” project (unless otherwise requested by the customer). Further, the interface to the customer is the same as for scenario B. There will be an additional activity, namely the one to interface the supplier of OBOSS.

Accordingly, any activities concerning the software management, possible to remove, are not identified.

Scenario B

All normal activities concerning the management of a software project of scenario B have to be executed.

Costs

By experiences from other “reuse” projects run by SES, no significant cost savings of management are expected when software is reused. Marginally, the number of hours per activity might be lower for scenario A than for B but SES has no figures showing that the amount of management work is significantly reduced in a “reuse” project.

3.7 Quality Assurance and Control

Scenario A

The same number of activities will be performed in this “reuse” project. However, for those activities “related” to reused units, the number of hours will be reduced.

Accordingly, any activities concerning the software quality assurance and control, possible to remove, are not identified but the total work is reduced.

See also §3.6 above concerning the development process.

Scenario B

All normal activities concerning the quality assurance and control of a software project of scenario B have to be executed.

Costs

The same number of quality assurance and control measures must be taken for scenario A as well as for scenario B to assure that the software will meet the required level of quality.

For scenario A, the cost savings of quality assurance and control are expected for those activities that are related directly to the reused software; e.g. inspections of code, test cases, and similar.



4. AN IMAGINARY USE CASE BASED ON SOHO COBS

4.1 Introduction

The order of saving achievable by using OBOSS in a software project has been investigated by evaluating the effect OBOSS would have had on the software development in a data handling system already developed. The software selected is the Central On-Board Software (COBS) of the Solar and Heliospheric Observatory (SOHO). The reason for selecting COBS for this imaginary OBOSS use case, is that it is a full-fledged on-board data handling software which has proved itself extremely reliable and flexible throughout the SOHO mission.

In order to arrive at a software project suited for a fair comparison, some project conditions had to be changed, e.g. the applicability of PUS and the availability of a compiler with which using OBOSS is feasible. Given the project conditions, the relative amount of software being subject to own, new development and OBOSS reuse respectively has been estimated for the two alternative scenarios of introducing OBOSS and not.

4.2 COBS Overview

This section gives a brief overview of COBS, its environment, tasks and implementation.

4.2.1 Context

The Central On-Board Software, COBS, constitutes the central data handling software of the SOHO¹ spacecraft, managing communications with ground and the instruments of the payload (Figure 2), along with numerous spacecraft monitoring and control functions. It was developed by SES during the first half of the 90's. It has since the SOHO launch on December 2, 1996 been updated with a number of patches and "trouble-shooting functions" to solve various spacecraft problems, the most recent being the addition of new monitoring functions for gyroless operations of the spacecraft.

¹ General information on the SOHO mission can be found at sohowww.estec.esa.nl.

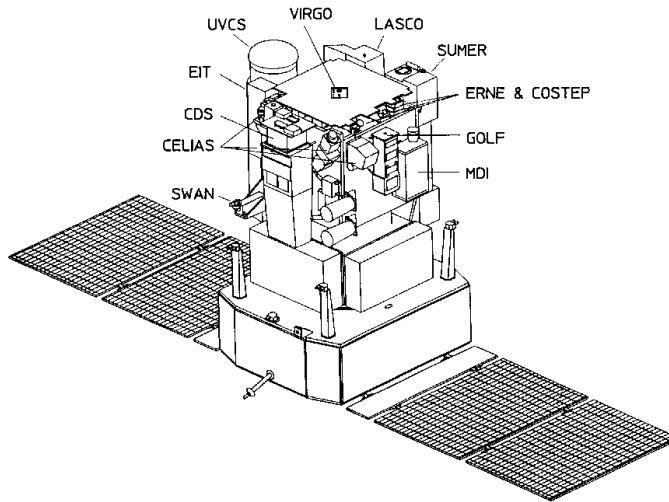


Figure 2 – SOHO and its instruments

Figure 3 shows the Data Handling Subsystem of SOHO. COBS resides in the Central Data Management Unit (CDMU), which connects to instruments, thermistors, heaters and so on within the satellite via three Remote Terminal Units (RTUs). Acquisition and commanding via the RTUs is performed by issuing interrogations on the On-Board Data Handling bus.

The CDMU may access the OBDH bus by single interrogations as well as by DMA controlled interrogation sequences, the latter of which is used for telemetry acquisition. The OBDH bus is thus used both for transferring data to/from other more or less intelligent sub-systems, and for performing device-level monitoring and controls, e.g. thermal regulation and power distribution control.

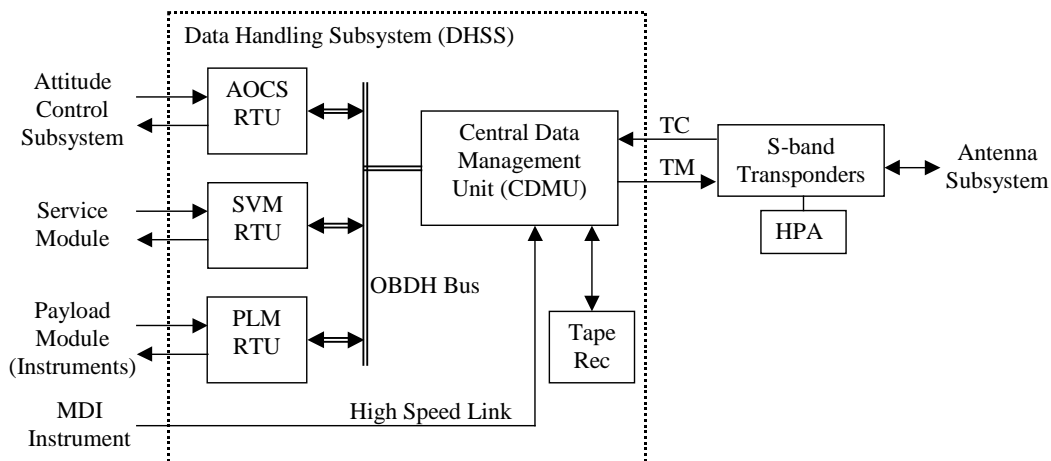


Figure 3 – The SOHO data handling subsystem (DHSS)



4.2.2 Tasks

COBS carries out a number tasks as summarised in the list below. Worth mentioning is also that with each monitoring function is normally connected a corrective action, performing various degree of reconfiguration and commanding within the spacecraft.

Basic functions:

- Software modes and start-up (boot, functions activation, data restore from context memory)
- DHSS surveillance and reconfiguration (functions for reliability/autonomy)
- Process management and scheduling
- Anomalies reporting
- Memory scrubbing
- Context memory management
- USO monitoring
- On-board time management and distribution
- Daily pulse distribution
- OBDH management
- OBDH block command management (data transfer protocol over OBDH bus)
- OBDH macro command and commands list management
- TM acquisition and generation
- TC management
- Power distribution unit (PDU) management
- Handle interface to various peripheral units accessed by application functions

Application functions:

- Min/max recording service
- Standard monitoring service
- Experiment (i.e. instrument) monitoring
- Emergency sun reacquisition (ESR) detection, warning message, and gyro setting
- Antenna pointing (HGA-APME) monitoring, control, and station-keeping
- Initial sun acquisition (ISA) and solar array deployment (SAD)
- Gyros monitoring
- Thermal control and monitoring
- Substitution heater monitoring
- Thruster monitoring
- RAAD monitoring
- Inter-instrument data exchange (solar co-ordinates)
- Memory management
- Delayed start-up
- Telecommand watchdog
- Trouble-shooting service (placeholder for uploaded trouble-shooting function)

Application functions uploaded after launch:

- Virgo (an instrument) cover open function (temporary)
- AOCS commanding unit (ACU) reset monitoring
- Reaction wheels monitoring
- Coarse Roll Pointing (CRP) and Hx momentum flags monitoring
- Experiments off-pointing warning message



4.2.3 Architecture and Scheduling

COBS is divided into about 120 software modules (Ada packages) organised into five architectural layers rising from hardware interface to application level functions.

The software processes are all cyclic, either at 38 Hz and high priority (mainly used for the basic functions), or at 1 Hz and low priority (used for application functions). Telemetry, for instance, is collected by setting up DMA controlled interrogation sequences every second 38 Hz-period, in parallel forwarding the data collected during the previous two periods to the telemetry frame formatting hardware, also by DMA. Application level processes are executed according to a ground controlled schedule, which is cyclic over 10 seconds, and with one second period as base.

4.2.4 Implementation

COBS is executing on a 1750A-based computer with 40 Kwords of PROM and 64 Kwords of RAM, a word being 16 bits wide. It is written in Ada, using the TLD compiler. The TLD runtime is cut to a minimum, mainly providing a number of library routines. Ada tasking is not used, neither are exceptions, generics nor variant records. This is partly due to performance constraints, partly because of deficiencies in the TLD compiler and runtime.

From an executional point of view, there are only two proper threads of individual execution. The simple process structure makes object protection superfluous in many cases, as its data are often accessed exclusively from within a single priority.

4.2.5 Development Model

The development of COBS was close to the ESA PSS-05-0 software engineering standard.

4.3 The Imaginary COBS Project

The conditions of the imaginary COBS project have been redefined to better match that of a first-time-PUS project of today. The following issues are identified, and are treated in the following subsections:

- TC/TM handling
- PUS applicability
- Compiler
- Target



4.3.1 TC/TM handling

The requirements on the TC and TM handling of SOHO must be assumed to follow another concept. The most obvious is that commands and reports shall comply to the PUS format, but also lower protocols and packet management have to be different as discussed in the following.

Some features of the TC handling in SOHO:

- ESA packet TC is not used, but a TC frame/block/message protocol stack.
- Low-level OBDH (device) commanding is provided already in the TC frame and TC message layers.
- An optional time-tag is provided already in the TC message layer.
- TC messages are immediately executed by COBS or transmitted to OBDH connected destinations (common handling through "OBDH block messages") in the order arrived. At most one TC per second is treated.

We assume that the ESA packet TC protocol is used instead, and that the device-level commanding and time-tag execution capabilities are taken care of by the corresponding PUS services (2 and 11). We still assume that the OBDH block message protocol is to be used for transmitting commands over the OBDH, but that there is required a gate in COBS to forward and format TC packets from the OBOSS packet router to the OBDH block messages driver.

Some features of the TM handling in SOHO:

- ESA packet TM is used, transmitted on two virtual channels. There are three different TM rates.
- Packet sizes are fixed for each APID.
- There are no event-generated packets, but periodic packets only according to a "TM format" consisting of 288 acquisition slots (generating zero or one packet) within a 15 seconds period. The science data part of the format is redefinable, while the housekeeping part is essentially not.
- Housekeeping as well as science data packets from sources connected to the OBDH bus are acquired according to the mentioned TM format and packetised by COBS. Different OBDH addresses (and different APIDs) are normally used for housekeeping and science data from an instrument. Lists of OBDH addresses define the housekeeping data collected directly by COBS.
- Some application functions use data in the acquired housekeeping packets, e.g. from the AOCs subsystem.
- COBS own "software packet" has a selectable second half with 18 variants. It contains status information from all COBS functions, TC execution acknowledges, anomaly reports, logs etc. Note that this packet is also transmitted periodically.

We assume that housekeeping and science data from other sources are either received over the OBDH bus as driven by "intelligent" sources (however polled by COBS as bus master), or acquired and packetised by reflecting application processes within COBS for "dumb" sources.



Furthermore, we assume a more flexible, PUS compliant housekeeping data collection for housekeeping packets collected directly by COBS, including the COBS internal parameters that constituted the COBS software packet. The fixed, periodical formats are thus abandoned in favour of event driven packets.

4.3.2 PUS Applicability

PUS is applicable to the imaginary project, which was not the case with COBS. Yet, COBS comprises equivalents to many of the PUS services, as shown in Table 1.

Service Type	PUS Service	OBOSS support	COBS equivalent exist	Remarks on the COBS functions
1	Telecommand Verification	✓	✓	Execution acknowledge/rejection reports, counters, error info, log
2	Device Level Commanding	✓	✓	Interrogation level commanding; single and list
3	HK & Diagnostic Reporting	✓	✓	Periodic, fixed report formats (as described above)
4	Parameter Statistics Reporting		✓	Min/max recording
5	Event Reporting	✓	✓	Anomaly reporting, anomalies log
6	Memory Management	✓	✓	Load, dump, checksum
7	Task Management			Execution profiles for LP processes are controllable
8	Function Management	✓	✓	Numerous application functions with related TCs and status information
9	Time Management			Adjustment and distribution within spacecraft is managed by COBS
10	Time Reporting		✓	Fixed period
11	Onboard Scheduling	✓	✓	Enbl/Dsbl, Insert/Delete
12	Onboard Monitoring	✓	✓	No validity parameter, but mask and a connected "macro" (OBDH interrogation list) to be executed
13	Large Data Transfer			
14	Packet Transmission Control		✓	Minimum capability for science data sources
15	Onboard Storage & Retrieval	✓		Tape recorders reside at TM stream level; logs are held on TC execution and anomalies
16	Onboard Traffic Management	✓	✓	TC and OBDH links status is provided; OBDH has "switchable channels" enabling ground to select between redundant OBDH addresses for some channels
17	Test		✓	Memory tests; XIO execution

Table 1 – Correspondence between PUS services and COBS functions



For the imaginary COBS project, we assume that the capabilities of COBS are retained (however defined in a PUS manner). Additionally, an extended capability is required for service types 1, 3 (regarding data collected by COBS), 11, 12 and 15 (to allow for TC verification and anomaly/event logging), to the extent covered by OBOSS.

4.3.3 Compiler

We assume the availability of a mature, stable Ada compiler. OBOSS uses Ada features such as generics, tasks, exceptions and variant records. COBS on the other hand used the TLD compiler without any of the mentioned features, and without the major part of the TLD runtime system. This was partly an early choice, but also a resort later in the project when the queerness of the TLD compiler eventually revealed. The Aonix to ERC32 compiler qualifies for being mature and stable, but is not well-optimising, which may cause performance and memory size problems (see next section). For COBS, it was performance considerations that forced us to abandon the Tartan compiler for the more optimising TLD compiler.

4.3.4 Target

Finally, we assume a target capable of executing the resulting COBS/OBOSS system without severe performance problems. This assumption actually neglects one of the major risks using OBOSS, but it is assumed that this risk has already been assessed and that the system is judged feasible.

4.4 Estimated Software Sizes

4.4.1 Model

In order to compare the amount of work required for developing the imaginary COBS/PUS system with and without OBOSS, a model of which software parts will undergo own development and which will be part of OBOSS has been outlined (Figure 4).

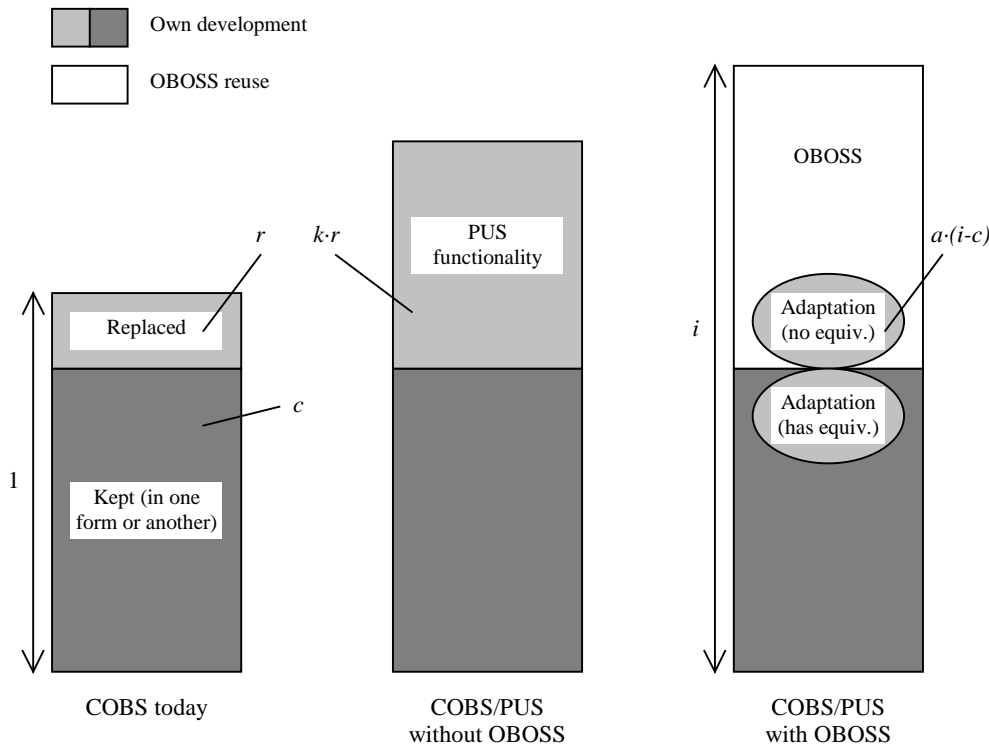


Figure 4 – Model used for estimating software sizes

The measures introduced in the model reflects the fact that only the ‘COBS today’ system is actually existent, and all other figures are more or less accurate guesses based on the available PUS and OBOSS information. The different properties are interpreted as follows:

- r* the amount of code replaced by other functionality when introducing PUS;
- c* the amount of code kept as is, or in other forms, e.g. OBOSS adaptations;
- k* a factor indicating the degree of extended functionality when PUS is introduced instead of the old, often simpler services;
- i* the inflation ratio when making a transition from old generation DHS to a new PUS compliant DHS utilising OBOSS;
- a* the relative part of the new code ($i - c$) that constitutes adaptations required when using OBOSS, but only the part that has no equivalent code when OBOSS is not used.

The code size measures are normalised to the size of COBS today ($r + c = 1$). For the COBS/PUS project, we get the following software sizes:

Own development, without OBOSS:	$c + k \cdot r$
Own development, if using OBOSS:	$c + a \cdot (i - c)$
OBOSS reuse, if using OBOSS:	$i - c - a \cdot (i - c) = (1 - a) \cdot (i - c)$



4.4.2 Kept and Replaced Parts of COBS

The code of COBS has been analysed and divided into three categories: replaced by OBOSS, partly replaced, and kept in one form or another. The term “kept” comprises all the code that would be the same in the two scenarios, as well as all the code that would still have to be developed when OBOSS is used, but in another form because the architecture different (we consider new development in both scenarios). The result is given in the following table:

Category	Lines of code ²	Share	Weight	<i>r</i>
Replaced	3052	13 %	1.0	0.13
Partly	3144	14 %	0.5	0.07
Kept	16995	73 %	0.0	0.00
Total	23191	100 %		0.20

Table 2 – Part of COBS kept and replaced

The parts replaced, in full or part, by OBOSS include (cp. 4.2.2) TC management, TM acquisition and generation, OBDH macrocommand and commands list management, anomalies reporting, standard monitoring service, memories management and functions activation control. The parts anyway developed mainly include computer core and DHSS surveillance functions, drivers and interfaces to near and far units, time-related functions, and application functions.

Using a 100-50-0% weighting, the amount of COBS code kept and replaced is estimated to be $c = 0.80$ and $r = 0.20$ respectively.

4.4.3 Adding PUS Functionality

There is not much to rely upon when estimating the amount of new code required implementing the new PUS functionality. Some of the services add substantially new functionality - viz. housekeeping & diagnostic reporting (service type 3), on-board scheduling (11) and on-board storage and retrieval (15) – while others are more or less of the same magnitude. PUS also adds another formatting layer (the data field header), and uses a more laborious parameter coding technique. Packet routing is more general.

Altogether, we estimate the increase in functionality to about $k = 3$. Note that this functionality is assumed to have been implemented in the same fashion the rest of COBS, i.e. an elaborated modularization, but without the ambitious general approach that comes with OBOSS.

² With lines of code is here ment Ada source lines containing code. Empty lines, lines with only a comment or pragma page are excluded.



4.4.4 Size of OBOSS and Related Adaptations

The resulting size of OBOSS, with all instantiations and necessary adaptations, is difficult to penetrate when the code is not available. An OBOSS case study is presented in [CaseStudy], which performed a similar transition from a non-PUS data handling system to a PUS-compliant one using OBOSS. The inflation ratio for the code size of the whole software system, normalised for the 16-bit to 32-bit transition in target system, is in the case study given to be 1.84. If the runtime system is excluded (not part of our model), the corresponding figure would become 1.49. If we also use the kernel services layer as normaliser (which should be functionally the same in both cases), we're back at 1.80. A value of 1.40 (given by Mr. Tullio Vardanega) is obtained from considering Ada statements. In the sequel, however, the value i will be used only for estimating the effort of unit testing and the size of adaptations, which suggests that we should consider the "post-instantiation" system rather than the "pre-instantiation" system.

As an estimate, we choose a value somewhere in between, say $i = 1.60$. Note that the value will only have a minor effect on the saving.

Is it then reasonable to assume that the OBOSS-supported implementation implies more code than a more targeted implementation? There are some indications that this might well be the case:

- An ambitious general implementation inevitably becomes larger than a specific one, simply because it solves a broader problem. Even if unused parts of the solution are possible to exclude, there is still some unavoidable structural functionality included, the very framework. On the other hand, a specific implementation will be chosen at the expense of less reusability, which is one corner stone of OBOSS.
- The way in which OBOSS instantiates application functions and PUS services entails a relatively large number of tasks and protected objects. A tailored implementation would probably host fewer tasks and interfacing objects.
- OBOSS brings about its own low-level support functionality. On the other hand, if appropriate, this support could be utilised also by the rest of the software, decreasing the amount of own development.

The amount of adaptation conditioned by OBOSS is equally difficult to estimate. The adaptations considered, are those being necessary to make the rest of the software fit with OBOSS, i.e. those parts that have no correspondence in a system without OBOSS. The major part of an application function like thermal regulation, for instance, would be present regardless of OBOSS, but might still need some particular adaptation to become connected with OBOSS. Another example is a "bus connector", which might need to bridge between the way OBOSS wishes to perform accesses and the reality, facing implementation constraints imposed by hardware or performance aspects (cp. DMA controlled OBDH sequences) - constraints that could be more easily managed by a more targeted implementation. A tentative estimate is $a = 10\%$.



4.4.5 Resulting Sizes

Applying the estimated figures into the model in Figure 4, we obtain an estimate regarding the amount of software expected to undergo own development and OBOSS reuse respectively for the two scenarios of using and not using OBOSS:

Kind of development	Original COBS	COBS/PUS		
		Without OBOSS	With OBOSS	Effect of OBOSS relative own dev.
Own development	1.00	1.40	0.88	-37%
OBOSS reuse	-	-	0.72	+51%

Table 3 – Amount of software developed

In short, OBOSS would be expected to reduce the amount of own developed software by 35-40% in our example project, but also itself contribute with somewhat more reused software.

4.4.6 On the Validity of the Result

It must be stressed that the obtained result is very sensitive to changes in the estimated variables, which in turn depend on the selected project conditions and the judgements of the authors. Nevertheless, we believe that this is how close we can get with the given time and information, pretty much like a real project situation when considering whether OBOSS should be introduced or not.

4.5 Saving in the Project

As discussed in chapter 3, it is primarily the design, coding and integration activities that experience a proportional cost reduction when reusing an already developed, presumably bug-free software component like OBOSS. This assumes no modification of OBOSS has to be done, which would otherwise rapidly hollow the possible saving.

For the unit testing, cost may decrease or even increase (due to larger and foreign software), depending on the provided support. With the appropriate support, much of the unit testing work for OBOSS with instances can be saved. In the table below we presume that the supplier has performed unit tests for non-generic modules, and that the test coverage is sufficient to fulfil the quality requirements of the project. In case it is necessary to re-execute tests on another target, compiler, or with other compiler options, we also must assume availability to the test programs. For generic packages, we presume that the same level of testing is performed and made available for a comprehensive instance, so it can be modified and executed for the actual instances. Assuming an equal mixture between the two kind of objects, together with some increase in size, we estimate the work for undertaking unit testing of the parts implemented by OBOSS to be about 40% of the work in the non-OBOSS scenario, i.e. a 60% reduction with the given premises.



No significant effect on requirements-analysis, system testing, or the development of integration and system tests is identified. Regarding management and QA activities, the major part is fixed activities (reviews, inspections etc.) as discussed in chapter 3.

The following table summarises the estimated saving that may be achieved in various project phases for software parts implemented by OBOSS. The division between phases is based on a number of actual projects, but may vary quite a lot for individual projects. Documentation and maintenance are not treated separately, but are included other relevant activities.

Activity	Part of project	Subject to cost reduction when using OBOSS	Saving for software parts implemented by OBOSS
Requirements analysis & definition	13%	0%	0%
Software design	18%	100%	18%
Coding	15%	100%	15%
Unit testing	11%	60%	7%
Integration and system test, development	17%	0%	0%
Integration and system test, execution	12%	50%	6%
Other (management, QA)	14%	30%	4%
Total	100%		50%

Table 4 – Saving for reused parts

Altogether, using OBOSS would allow for a 20% saving in the example COBS/PUS project, disregarding the cost of OBOSS itself. This is to be considered a ceiling for the COBS/PUS-system, as the estimated saving would be eroded by less suited project conditions.

4.6 Recurrent PUS Projects

The potential saving from using OBOSS compared to own development of PUS software naturally decreases as the own PUS platform gains functionality and maturity with recurrent PUS projects. The motive for saving time in recurrent PUS projects is rather the standard itself, not whether the corresponding software is procured or developed. One condition, of course, is that the benefit from adhering to a standard is well utilised by inter-project co-ordination within the company.



5. CONCLUSION

Cost savings are dependent on many factors: how well OBOSS covers the needed PUS functionality, the amount of new units to be developed, how much of OBOSS to be modified or adapted, availability of test software, the quality and availability of documentation, etc.

Some risks are identified when reusing OBOSS. Performance (size and time) could be difficult to obtain without any modifications of OBOSS. The use of advanced Ada features in OBOSS may be a problem when adapting the software to a certain platform. All possible problems can be difficult to foresee.

For the best benefit from OBOSS, the required PUS capabilities should be close to what is supported by OBOSS. This should be considered early in the system definition.

Design and Coding are probably those activities that contribute most to the cost savings when reusing software. Integration testing is also a candidate of significant cost savings, given certain prerequisites (e.g. documentation).

It is very important to define in detail all activities to be done when choosing the reuse approach. If activities associated with the reused software cannot be "reused" (tests for instance), the cost savings can be reduced. In such a situation, own development might even be cheaper than the reuse choice.

All prerequisites must, of course, be clearly expressed at a very early stage in a programme (in the invitation to tender, statement of work, proposal, customer's requirements, ground interface, etc.). This is necessary if a presumptive provider (contractor) of a data handling system shall be able to perform a relevant cost analysis.

Accordingly, to "maximise" cost savings, the following is valid:

- The customer must "harmonise" his specification to PUS and OBOSS.
- The activities required for the reused software must be clearly defined before the start of the project.

To build new COBS complying with PUS capabilities and harmonising with OBOSS, we have found the following:

In short, OBOSS would be expected to reduce the amount of own developed software by 35-40% of total software in our example project, but also itself contribute with somewhat more reused software.

Altogether, using OBOSS would allow for a 20% cost saving in the example COBS/PUS project, disregarding the cost of OBOSS itself. This is to be considered a ceiling for the COBS/PUS-system, as the estimated saving would be eroded by less suited project conditions.