AN OBJECT-ORIENTED COMPONENT-BASED FRAMEWORK FOR ON-BOARD SOFTWARE

Alessandro Pasetti[*], Wolfgang Pree[*], Jean-Loup Terraillon[**], Ton van Overbeek[**]

[*]Dept. of Computer Science, University of Constance, D-78457, Constance, Germany
[**]ESA-Estec, PO Box 299, 2200 AG Noordwijk, The Netherlands
pasetti@fmi.uni-konstanz.de, pree@acm.org, Ton.van.Overbeek@esa.int, Jean-Loup.Terraillon@esa.int

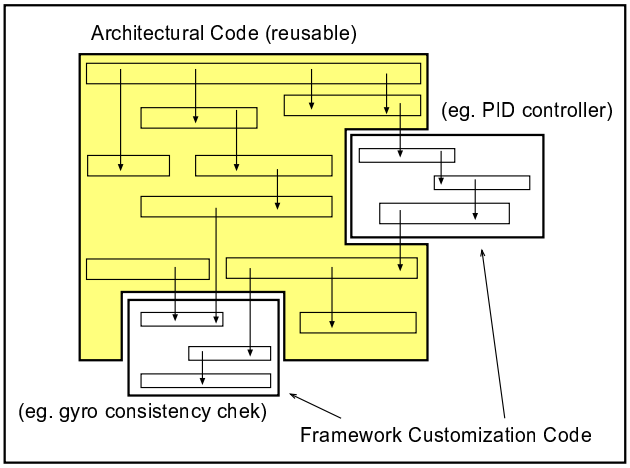
This paper will advocate the use of component-based software frameworks for on-board systems. Software frameworks are a reuse technology that makes *architectural* (as opposed to *code*) reuse possible. They have been successfully applied in the desktop and commercial arena. The sudden expansion in on-board memory and CPU resources subsequent to the space qualification of the ERC32 processor means that they can now be applied to space systems. This new technology was tested in a study done for ESA[1] that resulted in the development of a prototype component-based framework for the AOCS software. The framework was developed in C++ and tested on an ERC32 simulator. The experience of this study is that advanced software technologies can be safely applied to space systems bringing to them the same benefits as already seen in other fields.

The first part of the paper will present framework technology. The second part will describe the AOCS framework.

1. Component-Based Software Frameworks

Software frameworks promote the reuse of entire architectures within a narrowly defined application domain. They propose an architecture that is optimized for all applications within this domain and make its reuse across applications in the domain possible. Experienced software engineers who develop several related applications reuse architectures as a matter of course. Software frameworks are intended to formalize and make explicit this form of architectural reuse. They allow the investment that is made into designing the architecture of an application to be made available across projects and across design teams.

Frameworks can also be seen as *generative devices*. They serve as a basis from which applications can be rapidly instantiated. Indeed, in the commercial field, they are sometimes integrated in autocoding tools. Instantiation takes place at run time when the framework is configured to match the requirements of a particular application.



The figure at the side shows the structure of an application generated from a framework. The shaded block represents the architectural backbone of the application. This is invariant in the application's domain and is provided by the framework. The unshaded blocks provide the application-dependent behaviour. They customize the framework by being *plugged into* it during the instantiation process.

In practice, a framework offers three types of constructs to application developers:

[1] The views expressed in this paper are those of its authors only. They neither commit ESA nor reflect official ESA thinking.

- *abstract interfaces* or sets of related operations without implementation
- *design patterns* or optimized solutions to recurring design problems
- *concrete components* or binary entities implementing one or more interfaces that can be configured for use in a specific application at run-time

The design patterns encapsulate reusable architectural solutions and are therefore the vehicles through which architecture is made reusable. They additionally enforce architectural uniformity by ensuring that similar problems in different parts of the same application receive similar solutions. This endows the application architecture with a single "look & feel" that makes using and expanding it considerably easier.

The combination of components and design patterns means that a framework allows reuse not just of individual modules (subroutines, classes, etc) but of an entire system made up of the components together with their mutual relationships. The advantages of the framework approach stem from the fact it is precisely the definition of the system architecture that is often the toughest part of the software design process and it is the one that was neglected by previous reuse techniques.

2. Object-Oriented vs Object-Based Design

The paper will argue that, claims to the contrary notwithstanding, all space software developed to date is "object-based" rather than "object-oriented". By this it is meant that the software system is built as a collection of hierarchical objects that encapsulate data while exposing operations to handle them. The design process typically starts with an analysis of the problem domain to identify *concrete* classes and *concrete* objects. The resulting design is hierarchical in the sense that objects and classes are nested within each other.

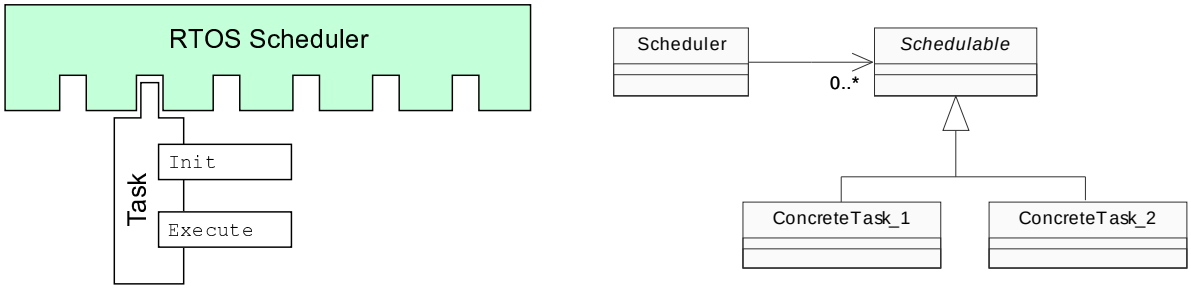
The truly object-oriented approach is radically different as it puts the emphasis not on objects, but on abstract interfaces. Design begins by identifying abstract functionalities that are then encapsulated in abstract interfaces. Objects are derived at a later stage of the design process as instances of classes that implement one or more abstract interfaces. Object-oriented architectures are hierarchical in the sense that they are based on hierarchical class trees with abstract classes at the top of the tree and concrete classes at the bottom.

The paper will argue that the current object-based approach allows software reuse at most at the level of code fragments. Architectural reuse – which is the level where reuse really begins to pay off – requires the availability of mechanisms to implement behaviour adaptation and to support design patterns. Use of this type of mechanisms is not contemplated by object-based design techniques (eg. HOOD) and not possible in object-based languages (eg. Ada83). Object-oriented methodologies and languages by contrast provide mechanisms like *object-composition* or *inheritance* that are specifically designed to implement behaviour adaptation. Software frameworks harness their power to make software reuse at the architectural level easy and quick.

3. The RTOS Example

The second part of the paper will discuss the architecture of the AOCS framework developed to demonstrate the application of object-oriented framework technology to a major satellite subsystem. Inspiration for its design was drawn from Real Time Operating Systems (RTOS's) that represent a highly successful (and often overlooked) example of software framework for real-time applications.

Consider for instance task scheduling which is a typical functionality offered by an RTOS. Firstly, an RTOS enforces a *design pattern* by assuming an application to be made up of tasks. Secondly, the RTOS separates the management of the tasks from their implementation and, at least conceptually, it does this by seeing tasks only through an *abstract interface*. Finally, the RTOS provides an application-independent, and hence reusable, *component* that encapsulates the management of the scheduling functionality. Thus, the RTOS offers all three constructs identified in section 1 and qualifies as a framework because the design pattern and the abstract interfaces are introduced to model adaptability and the component packages the invariant part of RTOS-based applications. The figure shows the RTOS reuse model for task scheduling:



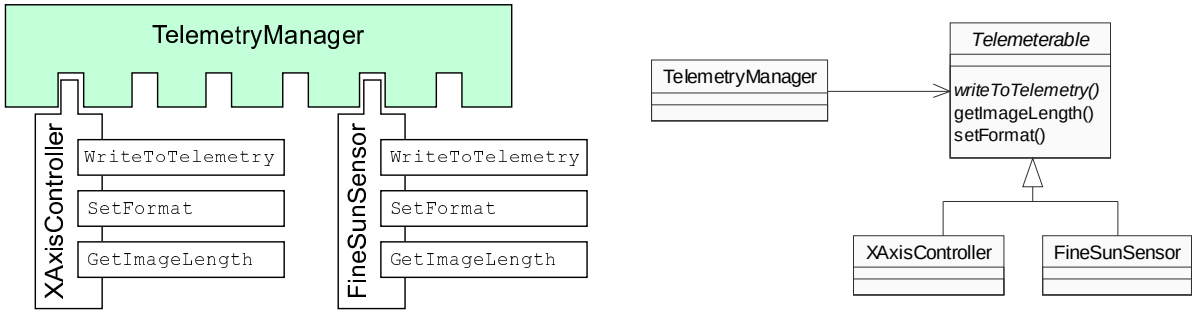
It is concluded that in an AOCS there is at least one functionality – task scheduling – where a framework approach improves reusability. Task scheduling is only one of many functionalities implemented by an AOCS and it is natural to ask whether the same principles that make it reusable could be applied to the other AOCS functionalities. The chief achievement of the AOCS project was to show that this is indeed possible.

4. The AOCS Framework Architecture

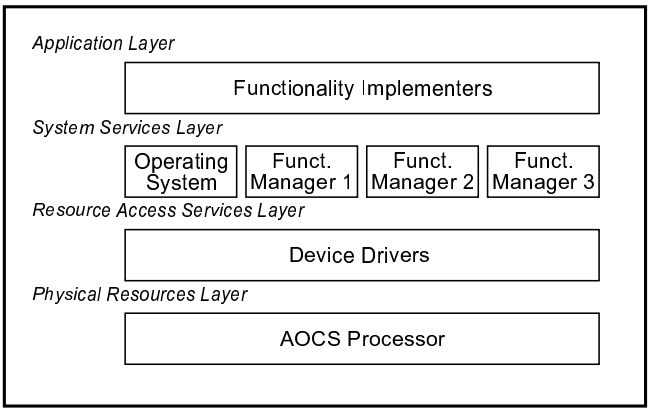
The AOCS framework design began by identifying the typical functionalities implemented by an AOCS:

- *Telemetry Functionality* (formatting and dispatching of telemetry data)
- *Telecommand Functionality* (management of telecommands)
- *Failure Detection Functionality* (management of checks to autonomously detect failures in the AOCS software)
- *Failure Recovery Functionality* (management of corrective measures to counteract detected failures)
- *Controller Functionality* (management of control algorithms for the control loops operated by the AOCS)
- *Manoeuvre Functionality* (management of manoeuvre like slews, wheel unloading, etc)
- *Reconfiguration Functionality* (management of unit reconfigurations)
- *Unit Functionality* (management of external sensors and actuators)

To each functionality the reuse model of the RTOS was applied. In the case of the telemetry functionality, for instance, the architectural solution is as sketched below. Objects whose state may potentially be included in telemetry are made to implement abstract interface `Telemeterable`. A telemetry manager component is then provided that manages a configurable list of objects of type `Telemeterable` and periodically goes through the list and asks each registered object to write its own state to the telemetry stream. Note the difference with the conventional approach where the telemetry handler has to manage a set of objects of disparate types and is responsible for collecting and packaging the telemetry data. These tasks are obviously application-specific and hence the telemetry handler cannot be reused.



The telemetry manager is one of many functionality managers offered by the AOCS framework. The figure at the side page shows the structure of an AOCS application instantiated from the AOCS framework. The framework can be seen as providing a domain-specific extension of the operating system in that it offers a number of functionality managers that are conceptually similar to operating system components in the sense that they provide an application-independent implementation of the management of recurring functionalities in the AOCS domain.



5. AOCS Framework Implementation

A prototype AOCS framework was developed in C++ for the ERC32 processor using the RTEMS operating system. Other object oriented languages might have been used. Ada95 was considered but discarded due to its poor support for multiple interface inheritance. The implementation took account of the real-time nature of the AOCS by ensuring that the worst case execution time of any code segment can be statically analyzed (for instance, no use was made of dynamic memory allocation or of run-time exceptions).

Use of a framework approach introduces some overheads. Since the framework was organized as an extension of the operating system, these overheads can be quantified. The memory overhead is given by the memory occupied by the functionality managers. For the AOCS framework prototype this was found to be just over 60 kBytes (including both code and data). Since a typical AOCS application might occupy around 400-600 Kbytes, the framework memory overhead is around 10-15%.

CPU overheads can be established by measuring CPU usage when the functionality managers run “empty” (ie. no components plugged into them). In the AOCS framework prototype, an empty cycle takes 0.2 ms with the SPARC processor running at 14 MHz. Since typical AOCS cycles last 100 ms or more, the framework-induced CPU overhead is nearly negligible.

6. Conclusions

The paper will terminate with a discussion of how the framework approach can be integrated in the ECSS-E40 process. It will be argued that the spirit, if not the letter, of the standard is difficult to reconcile with the framework approach notably because reuse at a level higher than individual component is ignored by ECSS-E40.

Finally, follow-up projects to the AOCS Framework Project will be briefly outlined. These include the use of the AOCS framework to generate part of the on-board software for the Proba satellite, the porting of the framework to real-time Java and the extension of its applicability to embedded control systems outside the satellite domain.