# A Framework for Embedded Control Systems: Methodological and Architectural Considerations

Alessandro Pasetti, Timothy Brown
*Faculty of Computer Science, University of Constance, D-78457, Constance, Germany,*
*e-mail: pasetti@fmi.uni-konstanz.de, tbb@acm.org*

Key words: Design Pattern, Software Framework, Framelet, Embedded Control System

Abstract: This paper presents a software framework for a satellite control system developed under a research contract for the European Space Agency. It is innovative in the methodology used to design it, in the architectural solutions it proposes, and in targeting a domain - that of embedded, mission-critical, real-time systems - that has been comparatively neglected in framework research. The paper begins by advocating the view that frameworks are artefacts offering three types of constructs to application developers: abstract interfaces, domain-specific design patterns and components. It then proceeds to describe the architecture of the framework. Inspiration for its design was drawn from an analysis of Real Time Operating Systems (RTOS) taken as models for reusable components in the embedded field. It is argued that satellite control systems can be conceptualised as collections of functionality managers similar to RTOS's and that the framework can be seen as a domain-specific extension to the operating system. This analysis of the framework architecture shows that design patterns and abstract interfaces, rather than concrete components, are the true foundation of a framework. This insight is used in the second part of the paper to propose a design process for frameworks that hinges on the division of the framework into so-called framelets. Framelets are units of design that simplify framework development by partitioning the space of design-patterns and abstract interfaces. They are to frameworks what subsystems are to individual applications. The experience from using the framelet approach in the satellite control system project is also discussed. The constraints imposed on the framework architecture by the real-time character of satellite systems are presented in the last part of the paper where it is shown how the framework separates the functional architecture from scheduling issues. The framework code and its detailed design documentation are available in a CD attached to this book.

# 1. INTRODUCTION

The resources, in terms of both memory and CPU power, available to embedded systems have traditionally been very limited severely constraining the complexity of their software that could rarely go beyond the modular paradigm typically implemented in C with generous sprinklings of assembler. As a result, despite the ubiquity of embedded processors – accounting for the overwhelming majority of processors in use – embedded software has traditionally remained the preserve of practitioners seldom attracting the attention of researchers.

This situation is slowly changing with the introduction of embedded processors with RISC architectures and multi-megabytes memory that are bringing to the embedded world the same resources normally taken for granted in desktop applications. Satellites are one class of embedded systems where the transition to high performance processors is taking place and they are therefore good candidates to test the applicability of advanced software technologies – such as software frameworks – to embedded systems.

Against this background, in 1999 a project was started under a research contract with the European Space Agency[1] (ESA) to develop a prototype software framework for the Attitude and Orbit Control System (AOCS) of satellites. AOCS's are real-time, mission-critical systems. They are often the most complex subsystem on board a satellite. Their structure – described in section 2 – is representative of that of embedded control systems in general and many of the considerations presented here are applicable to this broader category. The AOCS project is due to be completed in December 2000 and this paper reports its most significant results.

After discussing, in section 3, the definition of framework used in the project, the paper addresses the three main challenges that confronted the designers of the AOCS framework. The first challenge was architectural: the development of the AOCS framework required a reconceptualisation of the AOCS domain, an identification of the points where application adaptation was needed, and the definition of the corresponding adaptation mechanisms.

The approach followed in this project and described in section 4 is based on an analogy between frameworks and operating systems that leads to a view of frameworks as *domain-specific extensions to the operating system*. It is argued that, during the design process, this view helped suggest design solutions, that it now facilitates the description of the framework architecture, and that in the future it will make framework upgrades easier.

[1] ESA contract Estec/13776/99/NL/MV. The views expressed in this paper are those of its authors only. They do not in any way commit the European Space Agency or reflect official European Space Agency thinking.

The concept of *meta-pattern* is proposed in section 4.5 to formally capture the analogy between frameworks and operating systems and to describe the elements of large-scale uniformity in the architecture of the AOCS framework.

The second important challenge in the AOCS project was methodological. Whereas a wide range of methodologies exist to help application designers organize their software development process, corresponding methodologies for framework development have yet to emerge. This deficiency is very serious because frameworks are complex artefacts – necessarily more complex than the applications that are instantiated from them – and hence the need for a systematic approach to their development is all the more acutely felt.

In the AOCS project, one of the key methodological tools used to manage the complexity inherent in framework design were the so-called *framelets* which were introduced to break up the framework into simpler units. Framelets are also interesting from a theoretical point of view for the insights they provide into the nature of frameworks and because they throw into relief the conceptual differences between designing an individual application and designing a framework. Framelets are presented in section 5.

The real-time character of AOCS systems engendered the third and final challenge to the designers of its framework. This challenge was met with a two-pronged approach. On the one hand, language restrictions were imposed to ensure that the code of applications instantiated from the framework has timing properties that are statically predictable. On the other hand, the framework was designed to decouple architectural and scheduling issues and hence to make it possible to treat them as orthogonal dimensions. The real-time aspect of the framework is discussed in section 6.

Section 7 concludes the paper by considering the extent to which the methodological and architectural innovations introduced in the paper are applicable to other domains. It is argued that the framelet approach is applicable and useful in general because the way in which it divides a framework into smaller units reflects some fundamental properties of frameworks. The architectural solutions proposed for the AOCS framework – based on a meta-pattern and an analogy with operating systems – may be applicable to other embedded systems while many of the specific components and design patterns developed for the AOCS framework can be reused “as is” in other embedded control systems.

Finally, it should be noted that the source code and design documentation of the AOCS framework are available in the companion CD attached to this book. Hence, the objective of the paper is not to give an exhaustive description of the framework architecture. The emphasis is rather on the

general principles that inspired the framework design with particular attention to those whose relevance extends beyond the AOCS domain.

## 2. THE ATTITUDE AND ORBIT CONTROL SYSTEM (AOCS)

In order to put readers in the position to understand the material covered in the remainder of this paper, it is necessary to give a brief overview of AOCS systems (see [10,11] for a more detailed description).

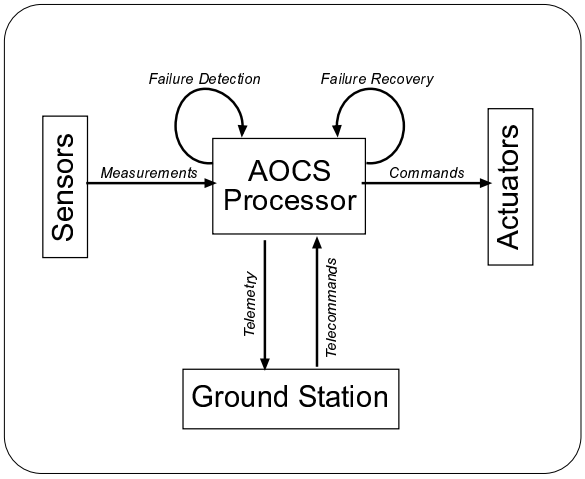



*Figure 1.* Structure of an AOCS System

The AOCS (see figure 1) is a typical embedded hard real-time control system. Its chief task is to periodically collect measurements from a set of sensors and convert them into commands for a set of actuators. The AOCS interacts with a ground control station from which it receives commands (*telecommands*) and to which it forwards housekeeping data (*telemetry*). Like all satellite systems, the AOCS must remain fully operational in the presence of any single failure and must survive prolonged periods of ground station outage. Robustness to faults and autonomy require the AOCS to perform failure detection and failure recovery actions.

From an implementation point of view, the AOCS software is normally organized as a set of statically defined tasks running under the control of a cyclical scheduler. Its code size typically lies between 15000 and 20000 lines of very compact code. This is expected to increase sharply in the future as more powerful space-qualified processors come into use.

## 3. WHAT IS A FRAMEWORK?

Software frameworks are a form of software reuse that primarily promotes 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 they make its reuse across applications in the domain possible [1,2,16,18,19,20 ].

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.

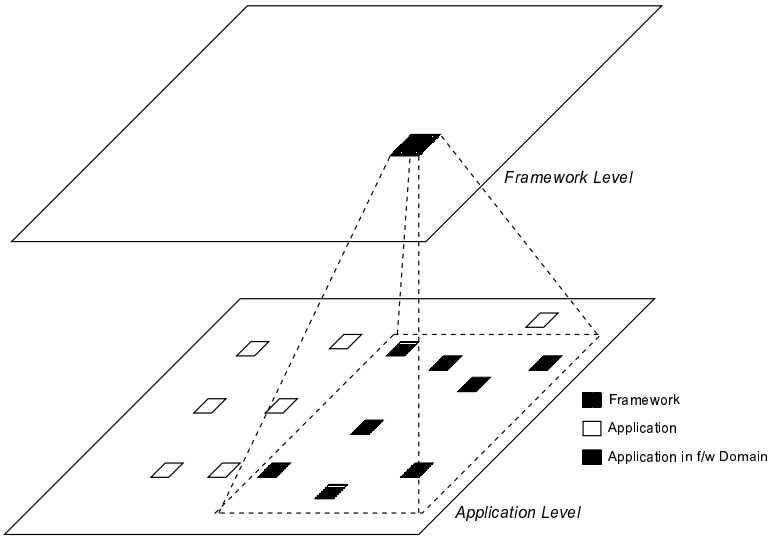



Figure 2: Framework and Application Levels

A framework therefore serves as the basis for the generation of a whole family of applications (see figure 2). The process whereby a concrete application is derived from the framework is called *framework instantiation*. The two key problems in framework design are to identify the points at which the framework behaviour is adapted to the requirements of a specific applications, the so-called *hot-spots* [13], and to devise ways to implement this adaptation. In this paper, frameworks are assumed to be object-oriented and the adaptation mechanisms are therefore based on abstract coupling (run-time adaptation) and inheritance (static adaptation).

The term “framework”, like so many others in computer science, is heavily overloaded and although most experts will agree with the definitions given above of what frameworks are for, there is as yet no consensus as to what exactly constitutes a framework. In the AOCS project, a framework was defined as consisting of:

– abstract interfaces
– design patterns
– concrete components

These are the constructs that the framework offers to application developers to help them build an application in the framework domain.

The *abstract interfaces* are declarations of sets of related operations for which no implementation is provided. Abstract interfaces capture behavioural signatures that are common to all applications in the framework domain. In a language like C++, they are implemented by pure virtual classes.

The *design patterns* are optimized solutions to recurring architectural problems in the framework domain [9]. Since they encapsulate reusable architectural solutions, and since frameworks are intended as vehicles for architectural reuse, design patterns are normally the heart of a framework.

Additionally, design patterns promote 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. Design uniformity is an important aspect of architectural reusability and a second important contribution of design patterns to frameworks.

The *concrete components* are binary units of reuse that implement one or more interfaces and that can be configured for use in a specific application at run-time. Framework components can be of two types: core components and default components. *Core components* encapsulate behaviours that are common to all applications in the framework domain. They are therefore used by all applications instantiated from the framework. *Default components* represent default implementations for some of the abstract interfaces in the framework. They encapsulate behaviours that are found in many applications in the framework domain but that are not intrinsic to the framework domain.

A framework will in general provide default components to implement the most common behaviours found in its domain. When very specific behaviours are required that are not catered to by the default components, application-specific components need to be created. These will have to conform to the framework interfaces and may often be obtained by specializing the default components through inheritance.

# 4. THE AOCS FRAMEWORK ARCHITECTURE

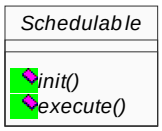
The primary objective of the AOCS project was to define a framework architecture for the AOCS domain. This task was accomplished in about six months. The resulting architecture is fully documented in the companion CD to this book. This paper therefore concentrates on presenting the general principles that inspired its design. In particular, an analogy with Real Time Operating Systems (RTOS) is developed that is interesting both because it provided the inspiration for the AOCS framework and because it points to a parallel between frameworks and operating systems.

## 4.1 The RTOS Example

One of the challenges of the AOCS project lay in the real-time and embedded characteristics of this type of applications which has seldom been the target of frameworks. In the initial phase of the project, however, it was realized that there is at least one highly successful example of frameworks for real-time systems. As will be shown in a moment, RTOS's fully fit the definition of framework given in section 3 and, because of their record in providing reusable software solutions, it was felt that they could serve as conceptual models for the AOCS framework. Accordingly, the AOCS project started with an attempt to answer the question: "How does an RTOS achieve reusability?". Three complementary answers were identified. They are illustrated below for the case of task scheduling which is a typical functionality offered by RTOS's.

Firstly, an RTOS enforces an *architectural infrastructure*. Namely, it assumes an application to be made up of tasks with certain well-defined characteristics (single entry point, single thread of execution, etc). Applications that wish to use the RTOS must conform to this architecture. Using the terminology of section 3, the RTOS proposes a design pattern.

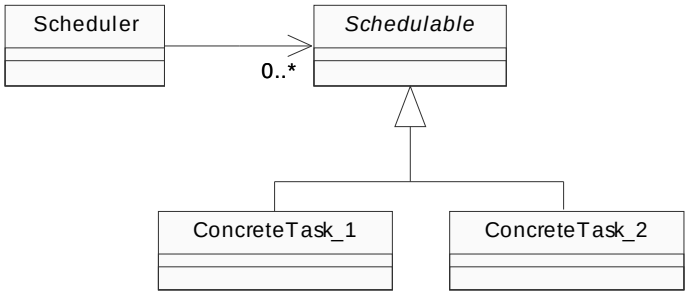
Secondly, the RTOS relies on the *separation of the management of a functionality from its implementation* and achieves this separation through an abstract interface. In the case of task scheduling, for instance, the RTOS sees a task as a component implementing the following abstract class:




where a call to `execute()` causes the task to initialize itself and a call to `run()` causes it to start executing. Separation through an abstract interface fosters reusability because it decouples the implementation of the

task, which is necessarily application-specific, from the scheduling policy which is used to activate it and which is application-independent.

Thirdly, the RTOS provides an application-independent, and hence reusable, component that encapsulates the management of the RTOS functionalities. The RTOS executable, for instance, will normally include a scheduler component that can be made application-independent because it sees the tasks it manages only through the abstract `Schedulable` interface:




In the terminology of section 3, the scheduler is a core component.

Thus, the RTOS offers all three frameworks constructs identified in section 3 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. It may be noted in passing that, conceptually, the primary entities are the design pattern and the schedulable abstract interface. The scheduler component is secondary as its function is essentially defined once the design pattern and abstract interface have been defined.

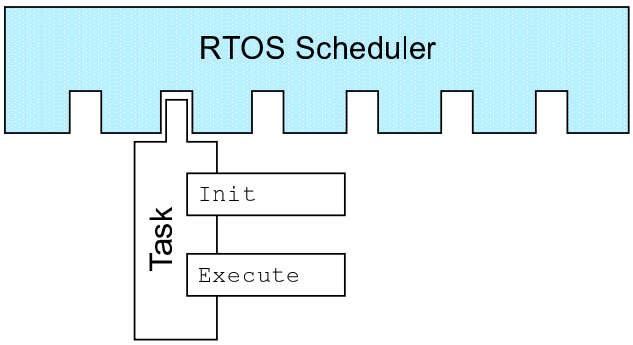



Figure 3: Plug-In View of an RTOS Scheduler

Figure 3 gives an alternative view of an RTOS-based system. The RTOS module is shown as a component exposing slots where application-dependent components can be plugged. The application-dependent components represent the tasks to be managed by the RTOS. The slots in the RTOS are therefore the hot-spots where the RTOS customisation takes place. The "shape" of the slots that determines whether a certain component can be plugged in is defined by the abstract interfaces that separate task management from task implementation.

## 4.2 The Lesson for the AOCS

One conclusion of the analysis in the previous section is that in an AOCS there is at least one functionality – task scheduling – where a framework approach can improve reusability. Task scheduling is of course only one of many functionalities implemented by an AOCS and it is therefore natural to ask whether the same principles that make scheduling management reusable could be applied to the other AOCS functionalities.

A large part of the design of the AOCS framework can be seen as an attempt to provide a positive answer to this question. This was done by first isolating the functionalities present in an AOCS and then systematically applying to each the reusability model of the RTOS.

The functionalities that were identified in an AOCS are:

- The *Telemetry Functionality* covering the formatting and dispatching of housekeeping information to the ground station.
- The *Telecommand Functionality* covering the processing and execution of commands received from the ground station.
- The *Failure Detection Functionality* covering the implementation of checks to autonomously detect failures in the AOCS software.
- The *Failure Recovery Functionality* covering the implementation of the corrective measures to counteract the failures reported by the failure detection functionality.
- The *Controller Functionality* covering the management and implementation of the control algorithms for the control loops operated by the AOCS.
- The *Reset Functionality* covering the control of the reset functions that bring the AOCS components to some initial default state.
- The *Configuration Functionality* covering the control of the configuration functions to clear all configuration information in the AOCS components, and to check that all components are correctly configured and ready to start normal operation.

– The *Unit Functionality* covering the acquisition of data from and the forwarding of commands to the external units managed by the AOCS (the sensors and the actuators).

From the point of view of the framework architecture designer, these functionalities can be treated as independent of each other. This is obvious in all cases with the possible exception of the failure detection and failure recovery functionalities that are sometimes merged together. In the AOCS framework, however, the failure detection manager constructs failure events encapsulating the description of any failures it has encountered and deposits them in shared repositories. In a separate activity, the failure recovery manager inspects the failure event repository and processes the failures according to their description.

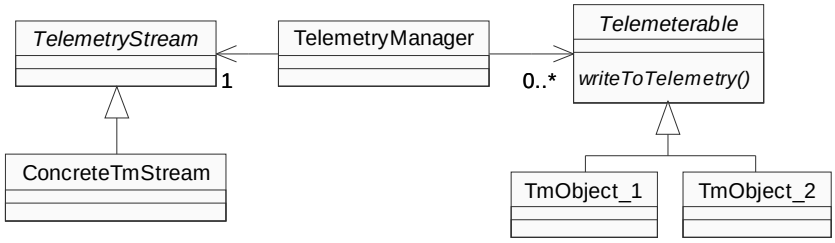
The mutual independence of the functionalities was crucial in simplifying the framework design as it allowed architectural solutions to be developed in isolation for each functionality. Architectural independence, however, did not degenerate into arbitrariness of solutions. As argued in section 4.5, design unity was maintained by imposing the same meta-pattern on all functionality management problems.

By way of illustration, the process that was followed in developing the architectural solutions to the functionality management problems will now be described for the case of the telemetry and controller functionalities.

## 4.3 The Telemetry Functionality

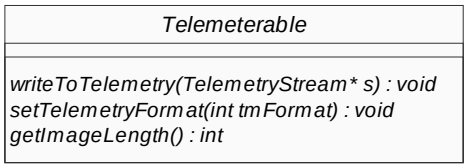
In current AOCS systems, telemetry processing is controlled by a so-called telemetry handler that directly collects telemetry data, formats and stores them in a dedicated buffer, and then has them transferred to the ground. To accomplish its task, the telemetry handler needs an intimate knowledge of the type and format of the telemetry data: it has to know which data, in which format, and from which objects have to be collected. It is this coupling between telemetry handler and telemetry data that makes the former application-specific and hinders its re-use.

In keeping with the RTOS reuse model, the approach taken in the AOCS project is based on a design pattern – the *telemetry design pattern* – that calls for a separation of telemetry management from the implementation of telemetry data collection. This is achieved by endowing selected components in the AOCS software with the capacity of writing themselves to the telemetry stream. Telemetry processing is then seen as the forwarding to the ground of an image of the internal state of some of the AOCS components. The resulting architecture is:



The ability of a component to write its own state to the telemetry stream is encapsulated in the abstract interface `Telemeterable` which must be implemented by all components intended for inclusion in telemetry. Its basic method is `writeToTelemetry`. Calling it causes a component to write its internal state to the telemetry stream. The telemetry manager is responsible for keeping track of the components whose state should be sent to the telemetry stream and for calling their `writeToTelemetry` methods to start the forwarding of telemetry data to the ground. The telemetry manager also needs to be aware of the data stream to which the telemetry data should be written. Since the hardware characteristics of the channel over which telemetry data are transmitted to the ground differs across AOCS systems, the telemetry stream is identified by an abstract interface, `TelemetryStream`. This interface defines the generic operations that can be performed on a data sink representing an ideal telemetry channel.

The second step in the development of an architectural solution for AOCS telemetry management was the instantiation of the telemetry pattern for the AOCS domain. This chiefly involved providing exact definitions for the `Telemeterable` and `TelemetryStream` interfaces. The former, for instance, was eventually defined as follows:




In the selected implementation, the telemetry stream is passed as an argument to method `writeToTelemetry` and methods are provided to check the size and set the format of the telemetry image generated by each component.

In the third and final step of the architecture design process, the telemetry manager component was characterized. The semantics of the telemetry pattern and of its associated abstract interfaces make its definition straightforward. The core of the telemetry manager component can be represented in pseudo-code as follows:

```
Telemeterable*      tmList[N_TM_OBJECTS];
TelemetryStream*    tmStream;
. . .
void activate()   {
  for (all TM objects ‘t’ in tmList)
  {  t->setTelemetryFormat(tmFormat);
     tmDataSize=t->getImageLength();
```

```
        if (object image fits in TM buffer)
          t->writeToTelemetry(tmStream);
        else
          . . .    // error!
      }
    }
```

Thus, the telemetry manager maintains a list of telemeterable components and, when it is activated, it calls their `writeToTelemetry` method. It uses the other operations exposed by `Telemeterable` to control the format of the telemetry images and to verify that the size of their image is consistent with the capacity of the telemetry channel. Additionally, and not shown in the pseudo code segment, the telemetry manager will offer operations to dynamically load and unload items from its internal list of telemeterable components and to set the telemetry stream at initialisation. Clearly, all these operations are application-independent and hence the telemetry manager is fully reusable or, in other words, it becomes a core component in the framework. The AOCS framework also provides a default component implementing the `TelemetryStream` interface for the case of a DMA-based telemetry channel that happens to be very common in AOCS systems.

The architectural solution to the telemetry management problem therefore exhibits the typical features of a framework being based on a design pattern that is specialized by abstract interfaces and a core component (the telemetry managers) and complemented by a default component (the default implementation of interface `TelemetryStream`).

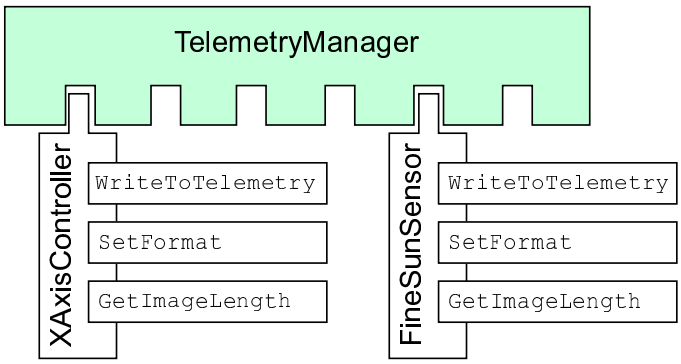



Figure 4: Plug-In View of a Telemetry Manager

Figure 4 shows a plug-in view of telemetry management. It again draws attention to the similarity between the telemetry management and the RTOS scheduling architectures and on how in both cases reusability originates in

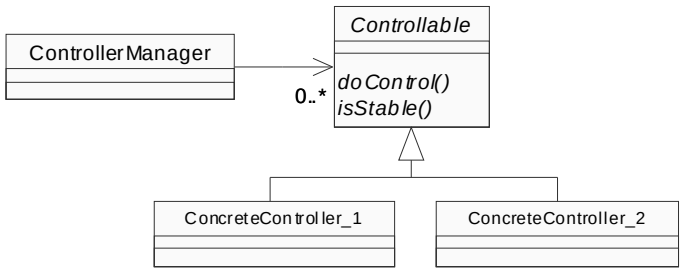
the introduction of an abstract interface to separate the management of a functionality from its implementation.

## 4.4 The Controller Functionality

An AOCS will typically contain several control loops serving such diverse purposes as stabilizing the attitude of the satellite, stabilizing its orbital position, controlling the execution of slews, or managing the satellite internal angular momentum.

The objects implementing these control loops tend to implement the same flow of activities starting with the acquisition and filtering of measurements from sensors, continuing with the computation, and ending with the application of commands to actuators designed to counteract any deviation of the variable under control (eg. the satellite attitude) from its desired value. Despite such similarities, existing AOCS's do not have a unified management of closed-loop controllers which tends to be entrusted to a multiplicity of one-of-a-kind objects that are each responsible for one single control loop.

The AOCS framework by contrast exploits the similarity among closed-loop controllers to apply to it the RTOS model of reusability. The *controller design pattern* is introduced for this purpose to separate the management of controllers from the implementation of the (application-specific) control algorithms:




As usual, separation is obtained through an abstract interface whose key method is `doControl` that directs the component to acquire the sensor measurements, derive discrepancies with the current set-point, and compute and apply the commands for the actuators. Since closed-loop controllers can become unstable, method `isStable` is provided to ask a controller to check its own stability.

The controller manager component is responsible for maintaining a list of objects of type `Controllable` and for asking them to check their stability and, if the stability is confirmed, to perform their allotted control action.

As in the case of the telemetry functionality, after the controller design pattern was identified, the second step in the architectural design was its instantiation for the AOCS framework. In this case, an abstract class `Controller` was used instead of interface `Controllable` since it was decided that there is some default behaviour that deserves to be encapsulated in a base class. Operations were also added to this class to set the controller in open and closed loop and to enforce limits on the actuator commands.

After the controllable design pattern has been instantiated and its associated class interfaces have been defined, the definition of the controller manager component follows naturally. Its core is:

```
Controller*  cnList[N_CN_OBJECTS];
. . .
void activate()   {
  for (all controllers ‘c’ in cnList)
  { if (c->isStable())
       c->doControl();
    else
       . . .    // error!
  }
}
```

Thus defined, the controller manager becomes a core component since its behaviour is completely application-independent. The AOCS framework offers some default components implementing commonly used control algorithms (eg. a PID controller).

The plug-in view of the controller manager is not shown but would be similar to that of the RTOS scheduler or of the telemetry manager with a controller manager that is customized with plug-in components of `Controller` type.

## 4.5 The Manager Meta-Pattern

The telemetry and controller design patterns are very similar. This is not coincidental since both were inspired by the same model of RTOS reusability. This commonality is so significant that it deserves to be captured in a new concept: the *manager meta-pattern*.

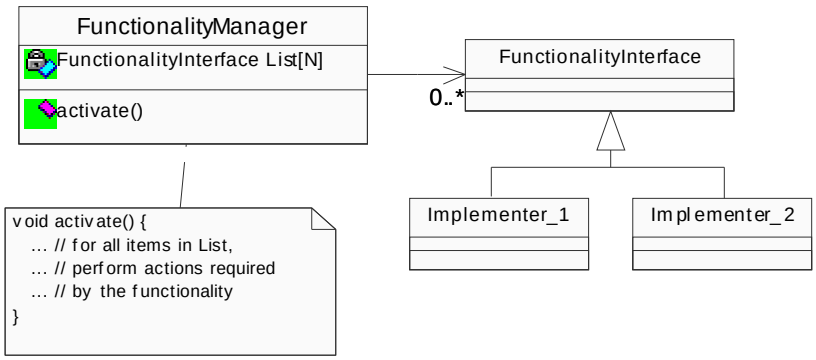
Design patterns define solutions for design problems at application level: they prescribe recipes that allow a class of related design problems confronting application developers to be solved in a uniform manner. The term meta-pattern is instead proposed to describe commonalities in different

design patterns used to address different design problems in the same framework. This usage is somewhat similar to that of [15]. Just as design patterns promote consistency at application level by ensuring that similar problems arising in different locations receive similar solutions, meta-patterns support design consistency at the framework level.

The manager meta-pattern addresses the problem of managing a functionality that requires the same actions to be repeatedly performed on a class of objects providing different implementations of the functionality itself. The solution it proposes can be summarized as follows:

– Define an abstract interface capturing the behavioural signature of the functionality. This interface separates the management of the functionality from its implementation.
– Build an application-independent and reusable functionality manager component (a core component). The functionality manager maintains a list of objects seen as instances of the functionality interface and it exposes an activation method that causes the actions required by the functionality management to be systematically performed on all objects in the list
– Where appropriate, provide default implementations of the functionality interface (default components)

The class diagram for the manager meta-pattern is:




The manager meta-pattern was applied to all the AOCS functionalities identified in section 4.2. This resulted in a framework architecture that is both elegant and homogeneous. Extensions and upgrades are facilitated by the mutual independence of the functionality managers that makes addition of facilities to handle new functionalities easy.

## 4.6 Overall Structure

Systematic application of the manager meta-pattern leads to a framework that will generate AOCS applications with an architecture as shown in figure 5. A collection of functionality managers represent the reusable architectural skeleton of the application. This is customized for a specific application by composition with components encapsulating the application-dependent functionalities implementations. To each functionality manager an abstract interface is associated.

The RTOS could be one of the functionality managers and the framework can therefore be seen as offering a *domain-specific extension to the operating system.*

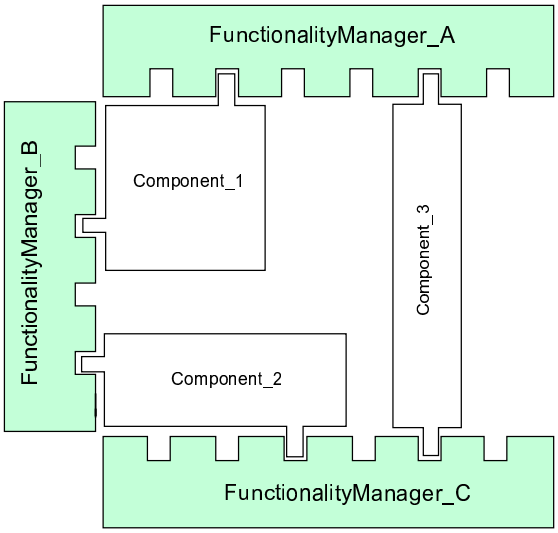



Figure 5: Structure of Framework-Generated AOCS

It should be stressed that the same component can be used to customize several functionality managers. A closed-loop controller, for instance, would obviously be plugged into the controller manager but might also be plugged into the telemetry manager if it is desired to include its state in the telemetry stream. This situation is handled through multiple inheritance with a component implementing all the interfaces associated to the functionality managers it customizes. Component_1 in figure 5, for instance, would have to implement (at least) the abstract interfaces corresponding to the functionality managers A and B.

Use of multiple inheritance is often resisted because it can lead to ambiguities [7] but the AOCS framework only requires multiple inheritance

of abstract interfaces which is known to be safe. This kind of multiple inheritance is used extensively throughout the AOCS framework with typical AOCS components normally implementing 5 or 6 different abstract interfaces each representing a different functionality.

Figure 6 shows how the application architecture implied by the AOCS framework can be recast according to the Generic Open Architecture (GOA) model [8]. GOA is a reference architecture for embedded software to provide a foundation for effective open systems. It distinguishes four layers in a software system: application software, system services, resource access services, and physical resources. In a GOA perspective, an AOCS application is most naturally decomposed by placing the functionality managers in the system services layer alongside the operating system. This choice is dictated by the application-invariant character of the functionality managers and by the fact that they – unlike the components implementing the functionalities – need access to the hardware drivers in the resource access layer. The telemetry manager for instance has an interface to the hardware channel through which telemetry data are routed to the ground.

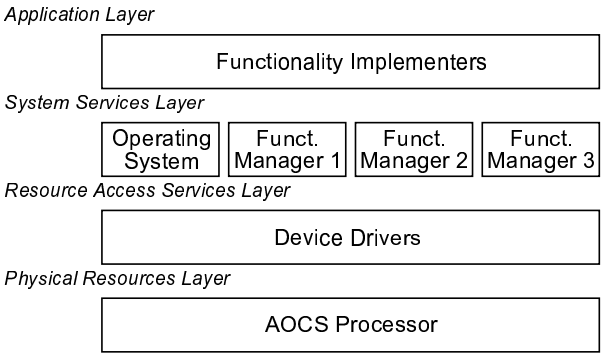



Figure 6: GOA Model of the AOCS Architecture

Figure 6 illustrates again the affinity of functionality managers and operating systems and shows that the framework factors out of an application OS-like functionalities encapsulating them into application-independent modules.

## 4.7 Framework Overheads

The question of the overheads introduced by using a framework to develop an application is often asked but quantitative answers can seldom be provided because the framework may consist of many components only

some of which will be used in a specific application and because it is difficult to trace an application's thread of execution in and out of the framework elements. With the approach used in the AOCS framework, however, the framework overheads can be measured in the same manner in which OS overheads are usually measured.

The memory overhead is simply given by the memory occupied by the functionality managers which represents the ballast that the framework introduces at application level. With the configuration used for the AOCS framework prototype (SPARC processor with GNU C++ compiler) this is equal to 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.

## 4.8 Architectural Infrastructure

The functionality managers address localized design issues in the AOCS domain but do not exhaust the framework since they cannot handle non-localized infrastructure problems. Some of these problems are outlined below together with the solutions proposed by the AOCS framework.

One first example of infrastructure problem concerns inter-component communication. The architecture promoted by the AOCS framework is component-based. Some mechanism is required to allow these components to exchange information. The solution offered by the AOCS framework was inspired by the JavaSpaces concept and is built around a design pattern to allow components to retrieve data items from shared repositories.

A second example of infrastructure problem arises from the need of components to monitor each other both in order to synchronize their behaviour and for purposes of failure detection. The framework addresses this problem by means of three distinct design patterns loosely based on the property mechanism of the JavaBeans architecture.

AOCS components must normally adapt their behaviour to their environment. The control algorithm to be implemented by a closed-loop controller, for instance, when the satellite is tumbling at a high angular rate is very different from that which is required when the satellite is nearly stationary. This dependency on the external environment is captured by making components mode-dependent. Mode-dependency in the AOCS framework is implemented using a variant of the strategy pattern [9].

Other infrastructural problems in AOCS applications concern the handling of unit reconfiguration and the handling of sequential data processing chains. For both, the AOCS framework offers dedicated domain-specific design patterns.

For more details on how infrastructure problems are treated, readers should refer to the attached project documentation. The point to note here is that their solutions – like the solutions to the functionality management problems – are based on the definition of domain-specific design patterns and of the abstract interfaces required to specialize them. In many cases, core and default components were also developed but in the framework design process they always came second to the design patterns and abstract interfaces.

## 5. THE FRAMELET CONCEPT

The previous section reconstructed the thought processes that led to the design of the AOCS framework. This type of analysis is valuable because it might hint at general rules to be incorporated into a framework design methodology. Framelets were one of the methodological concepts whose evolution was stimulated by the AOCS project.

One observation to emerge from the approach delineated above is that design patterns and abstract interfaces are the basis for framework design. Framelets formalize this insight and provide a means to address the quantitative aspect of framework complexity. Quantitative complexity in frameworks stems from their sheer size: a single framework may encompass hundreds of constructs – classes, design patterns, interfaces, hot-spots, etc – embedded in an often tangled web of interconnections and semantic relationships. Framelets are introduced to simplify the design and the description of a framework by allowing it to be broken up into smaller and simpler entities.

The need to tackle quantitative complexity is of course not restricted to frameworks. It also arises in the case of conventional application design where the classical solution is to subdivide the application into *subsystems* that partition the space of components making up the application. The subsystems are designed to be as self-contained as possible and to have minimal coupling. This makes it possible, to some extent, to develop them independently of each other thus reducing design complexity.

This approach works well for individual applications that are just the sum of their components and, as initially proposed in [3, 22], the framelet concept mirrored it in simply subdividing the set of components offered by a

frameworks into smaller subsets and treating framelets essentially as "small frameworks".

However, frameworks consist of more than just components for they also include design patterns and abstract interfaces among their basic constructs. In [12], therefore, framelets were redefined to designate sets of logically related components *and* design patterns and interfaces. It is now felt that this view too must be abandoned because, in the case of frameworks, components take second place to design patterns and interfaces that are the true foundations upon which frameworks are built. The behaviours implemented by the core components are normally implied by the design patterns and are therefore conceptually subordinate to them while default components are simply implementations of abstract interfaces.

Given these relative priorities among the constructs making up a framework, the transposition of the subsystem approach to framework design requires a shift of focus from components to abstract interfaces and design patterns. Simplification in other words must be achieved by partitioning not the space of components, but that of design patterns and abstract interfaces. The framelet term was introduced to designate non-overlapping groups of logically related design patterns and interfaces. The two ways of partitioning the design space – through framelets and through subsystems – are schematically contrasted in figure 7.

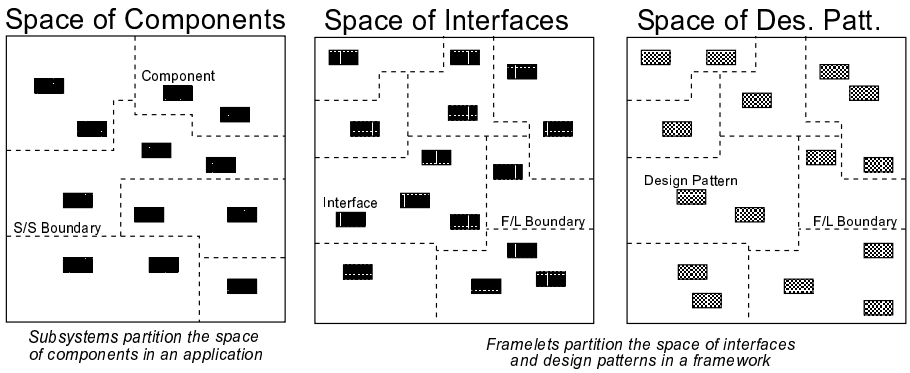



Figure 7: Design Simplification in Frameworks and Applications

It is important to stress that there is no guarantee that framework components can be mapped to single framelets. Indeed, in most cases they will implement several interfaces possibly belonging to different framelets and will participate in several design patterns possibly provided by different framelets. The framelet structure cannot therefore be imposed upon the space of components each of which will normally straddle framelet boundaries. In

the AOCS project, for instance, virtually all components offered by the framework intersect multiple framelets simultaneously.

Conversely, it is normally possible to partition the space of abstract interfaces and design patterns into homologous subsets because the abstract interfaces exist to support the variability and adaptability introduced by the design patterns and therefore abstract interfaces can normally be unambiguously associated to design patterns.

Hot-spots are not primitive constructs in a framework because they are reducible to design patterns and abstract interfaces but it is interesting to note that, precisely for this reason, hot-spots can be assigned to framelets. Framelets, therefore, can also be defined as a partitioning of a framework into subsets of logically related abstract interfaces, design patterns and hot-spots.

In practice, logical relationship means that each framelet addresses a specific design problem in the framework. Framelets in other words become *units of design* with the framework being obtained by combining the framelets.

Some of the characteristic features of framelets that follow from all the above are:

– *Small size:* since they are made up of closely related constructs, framelets will typically be small. In the AOCS framework, framelets consist of 2-3 design patterns and a handful of abstract interfaces.
– *No execution control assumptions*: since they are intended to be integrated with each other to form the framework, framelets must not assume that they have control of the application execution.
– *Self-standing*: since they address a specific design problem, framelets will normally be self-standing in the sense that they could be used in isolation from the other framelets (perhaps in a different framework).

It is noteworthy that all the above features contribute to simplifying design and justify the claim that framelets help reduce the complexity of the framework design process.

Framelets make two additional contributions to frameworks. Firstly, they simplify the *extension* of a framework since they make it possible to add new functionalities to it by adding new framelets. Secondly, a framelet-based framework is easier to compose with other frameworks since it does not assume that it has control of program execution and this assumption is a common obstacle to framework integration [4].

## 5.1 Implications for Design Process

Adoption of a framelet-based approach to design a framework has profound implications on the framework design process. Most design methodologies (see for instance [5,14]) begin the design process with the identification of the basic concrete objects in an application. These are then subdivided into simpler objects thus resulting in a system that is hierarchical in the sense that its objects and classes are nested within each other. This way of proceeding is the natural one when complexity is mastered by partitioning the space of components or modules.

If, however, complexity management is done at the level of abstract interfaces and design patterns, as is the case with framelets, then the attention must shift to the interfaces and the patterns. The design process must start by identifying abstract functionalities that are then encapsulated in abstract interfaces, and points of framework adaptability to which design patterns are fitted. Concrete classes and the components instantiated from them are introduced only at a later stage as implementation vehicles for the interfaces and the patterns.

The resulting architecture will again be hierarchical but in the different sense of being based on hierarchical a class tree with abstract classes at the top of the tree and concrete classes at the bottom.

## 5.2 Framelets in the AOCS Framework

The AOCS framework served as a test bed for the framelet concept. The framework was developed from the start as a collection of framelets. Framelets were built around *atomic design problems* representing the lowest level of design problems for which an architectural solution can be defined in isolation from the rest of the framework.

In practice, each functionality manager and each infrastructure problem gave rise to a framelet. The mapping from functionality management and infrastructure problems to framelets is consistent with the characteristics of framelets as described above. First and foremost, it leads to a partitioning of the space of design patterns and abstract interfaces because each functionality management and infrastructure problem gives rise to a small number of dedicated design patterns and abstract interfaces.

Secondly, the resulting framelets are small in size typically requiring only a handful of basic constructs.

Thirdly, the AOCS framelets do not make any assumptions about execution control. This is certainly the case for the framelets covering the infrastructure problems but functionality management framelets would seem to assume some kind of execution control model since the functionality

managers are active components which might imply some execution control model. In fact, one of the key features of the AOCS frameworks is the decoupling between architecture and scheduling as discussed in section 6.

Fourthly, the AOCS framelets are self-standing in the sense that the basic constructs of each framelet – its abstract interfaces and design patterns –are defined independently of those of other framelets and in the sense that they are in principle usable in contexts other than the AOCS framework. The telemetry framelet, built around the telemetry functionality manager is representative because the design solution it offers would be reusable in any other embedded system where housekeeping data must be regularly sent to an operator or external monitor.

The AOCS project undoubtedly benefited from the framelet approach. The framework design became more manageable as each framelet covered a design problem that could be handled independently from other design problems. Indeed, in some cases, framelets were allocated to different designers working in parallel. Reliance on framelets to decompose and simplify the framework was therefore instrumental in reducing design time and in the future it will certainly ease both its maintenance and upgrade.

## 5.3 Framelets and Aspects

It should again be stressed that the framelets do not partition the space of components. Consider for instance a controller component (see section 4.4). This component, as already noted in section 4.6, might implement two different abstract interfaces exported by the controller and telemetry framelets. Thus, it can be said that the component *participates* in both the controller framelet and the telemetry framelet. Additionally, since the controller needs to exchange data with other components, it will make use of the infrastructure design patterns (see section 4.8) and thus participate in the framelets which specify those patterns. Thus, this single component straddles framelet boundaries and cannot be assigned to any one framelet.

In the terminology used by the literature of Aspect-Oriented Programming [23,24] a feature of a software system that affects, could potentially affect, or requires the cooperation of multiple classes in the system is said to *cross-cut* the classes. To state it more generically, a feature or behaviour which cannot be readily encapsulated in one modular unit (subroutine, procedure, module, class, etc.) offered by the programming language or design technique being used is said to cross-cut the modular units. Such cross-cutting features are referred to as *aspects*.

Using this terminology, it can be said, for example, that being telemeterable (as defined by implementing the `Telemeterable` interface)

is one aspect of the controller component mentioned above. Being a controller (as defined by being a subclass of the abstract base class `Controller`) is another aspect of the controller component. Other aspects of the controller component's behaviour will be defined by the component's participation in other framelets from the AOCS framework design.

Thus each framelet provides the definition, via its design patterns and abstract interfaces, of how any given component should implement certain aspects of its overall behaviour. In this sense, it can be stated that framelets are an *Aspect-Oriented Design* technique for frameworks.

As such, the framelet approach simplifies the *framework design* process but does not as significantly simplify the framework *implementation*. The basic modular unit of implementation in any object-oriented system is the class. Since the behaviours defined in a framelet cannot be delegated to one specific class, they will need to be implemented separately in the various classes which make up the framework. For example, all components in the framework or an application instantiated from the framework which wish to add the telemeterable aspect to their behaviour will need to have their own implementation of the methods which make up the Telemeterable interface.

The implementation of the telemeterable aspect may be very similar across classes. Thus there is some duplication of effort in the implementation of the various classes. Also, if the `Telemeterable` interface changes, there will be several places throughout the implementation which will have to be changed to accommodate the changes in the `Telemeterable` interface. Discovering and factoring out those similarities into a separate, single implementation is one of the goals of the research in aspect-oriented programming.[2] Thus aspect-oriented programming's goal is to simplify the implementation just as framelets simplify the design.

## 6. THE REAL-TIME DIMENSION

AOCS systems must be *demonstrably schedulable*. An application is schedulable if it can meet all its deadlines under any operational conditions. The schedulability requirement derives from the hard real-time nature of the AOCS. Additionally, since AOCS systems are mission-critical,

[2] The separate implementation of an aspect is typically done in a different language than the language used to implement the basic components of the system. Thus an aspect-oriented implementation will be done partially in a *component language*, like C++ or Java, and partially in a set of specially designed *aspect languages*. The code from both the component language and the aspect languages is then *weaved* together to form a single application program using a tool called an *aspect weaver*.

schedulability must be demonstrable in the sense that it must be possible to verify it statically by analysing the source code. This section discusses the impact that these requirements have on the AOCS framework.

## 6.1 Execution Time Predictability

A necessary condition for the demonstrable schedulability of a software application is that the execution time of any of its code segments be statically predictable. This requirement has a repercussion at framework level insofar as some of the code in an AOCS application is inherited from the framework (core and default components).

The AOCS framework safeguards timing predictability primarily by avoiding constructs and operations that lead to non-predictable code. In practice, this led to the avoidance of exception handling and dynamic memory allocation. The former presented no particular problem as the framework infrastructure offers a mechanism for reporting events that can be used *in lieu* of exceptions. The latter was instead rather difficult to achieve. Most existing frameworks are targeted at non-real-time applications and make extensive use of dynamic object creation and destruction. Avoiding such operations required some rethinking of typical design patterns used in frameworks and sometimes resulted in slightly awkward constructs.

In the AOCS framework, all objects required by an application are created statically – by a set of abstract factories – during initialisation and are never destroyed afterwards. Components operate upon and exchange objects exclusively through pointers. The ban on object destruction is enforced by introducing a root class from which all non-trivial framework classes are derived and by making its generate an error which ensures that any inadvertent object destruction is flagged. Obviously, this prevents object creation and destruction on the stack as well as on the heap. This is more restrictive than needed for timing predictability but it has the advantage of removing the problem of dangling pointers.

As already mentioned, the AOCS framework relies extensively on dynamic binding to model application adaptability. At first sight dynamic binding might seem to be incompatible with code execution predictability because of the impossibility of statically associating a definite piece of code to a particular method call. However, in embedded systems there is no dynamic class loading and hence the number of methods that *might* be called is finite (and often small). It is therefore always possible to determine worst-case execution times and use this estimate in the schedulability analysis.

This type of estimate is likely to be pessimistic but pessimism is a feature of *all* methods for static schedulability analysis. Note moreover that run-time

binding is often used as a substitute for conditional branches and therefore the conventional and object-oriented solutions will often lead to identically pessimistic timing execution estimates.

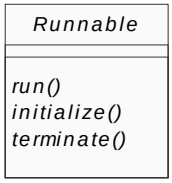
The heavy reliance of the framework on design patterns poses a more serious problem since some design patterns have a recursive structure [6] that makes static timing analysis impossible. In such cases, timing predictability can only be retained by adding meta-information to the source code that specifies the maximum depth of the recursion. In order to make it easy to provide this information, the framework documentation identifies all recursive design patterns that may be introduced in an AOCS application.

Upper bounds on the depth of recursion can be inferred from the design pattern semantics. Consider, for instance, the handling of failure reports which in the AOCS framework is done using the chain of responsibility pattern [9]. This pattern is recursive because it can link together an unlimited number of handlers. Any specific AOCS application, however, will have a fixed number of handlers for the failure reports and this will bound the recursion implied by the chain of responsibility pattern.

## 6.2 Compatibility with Scheduling Policies

Execution time estimates are useful to verify schedulability only within the context of a specific scheduling policy. In order to preserve its generality, the AOCS framework attempts, as far as possible, to make the software architecture independent of the scheduling policy.

As discussed in section 4, an AOCS application is seen as a bundle of functionalities. The framework assumes that to each functionality a thread of execution is associated. For this purpose, the functionality managers are made to implement the following interface:




The key method is `run`. When it is called, the functionality manager performs all the actions associated to the current cycle of operation. Method `run`, however, can only be called from outside the framework (normally by an external scheduler or by an interrupt servicing routine). Additionally, the functionality managers are designed not to make any assumptions about the frequency with which their method `run` is called or about the source of the call. Whenever they are activated by a call to method `run`, the functionality managers take whatever action is appropriate at the time they are called

without regard to how recently they were last called in the past or to how soon they may expect to be called again in the future. This insulates them – and hence the framework – from the AOCS scheduling policy.

Obviously, the components that implement the AOCS functionalities and that customize the functionality managers will rely on their methods being invoked according to some timing pattern but these components are application-specific and are defined and configured by the application developer at application-level. Their dependency on a particular scheduling policy therefore does not carry over to the framework.

As an example consider the controller functionality described in section 4.4. A typical concrete controller component will normally implement a digital filter and therefore, in order to work properly, it will need to have its internal state propagated at regular intervals. Such a component will therefore have a built-in assumption about the frequency with which some of its methods must be called. No such assumption, however, applies to the controller manager component that sees the concrete controllers through an abstract interface whose operations do not imply any timing requirements.

The framework architecture therefore achieves independence from the scheduling policy of the AOCS application by confining scheduling assumptions to the application-specific components.

In one notable case, this independence was concretely verified by demonstrating that the AOCS framework is compatible with HRT-HOOD. HRT-HOOD is a modification of the HOOD methodology that is based on a particular scheduling model and that is designed to produce systems that can be statically analyzed for their timing properties [14]. As an object-based methodology, it cannot be directly used to design framework-based – and hence object-oriented – AOCS applications. It was, however, possible to show that an application instantiated from the AOCS framework could in principle have been produced using HRT-HOOD. This is an important result since it guarantees that the framework generates applications whose schedulability can be statically analysable.

It should finally be noted that some scheduling policies require the implementation of synchronization mechanisms to coordinate access to shared resources. The framework regards any public method as potentially giving access to a shared resource and therefore as potentially in need of an access synchronization mechanism. The type of mechanism to be used, however, is treated as an implementation rather than an architectural issue and is therefore neither provided nor dictated by the framework.

# 7. CONCLUSIONS

This paper has touched upon both architectural and methodological issues. The discussion was largely done in the context of the AOCS project, it is therefore appropriate to ask to what extent its conclusions are more widely applicable to other domains.

The common thread to the concepts presented in the paper is a view of frameworks that gives privileged status to design patterns and abstract interfaces. This is in contrasts to individual applications where the basic architectural construct are the concrete classes and the components instantiated from them. The value of this view is quite general because it is rooted in the different purposes served by frameworks and applications. The development of the latter is normally driven by very specific needs and the key problem then is the modelling of the behaviours that are induced by these needs. Concrete classes become the focus of design attention because they are the most natural vehicle to encapsulate the implementation of behaviours. Frameworks on the other hand are created to model adaptability and adaptability is best modelled by design patterns and abstract interfaces which accordingly become the basis of framework design. Frameworks also offer concrete components but their definition is conceptually subordinate to that of the design patterns and abstract interfaces.

It follows from the above that just as design simplification in the case of applications is achieved by partitioning the space of concrete classes into subsystems, in the case of framework the same objective is achieved by partitioning the space of design patterns and abstract interfaces. The concept of framelet was introduced in response to this need to designate a set of logically related design patterns and interfaces that address a design problem in isolation from the rest of the framework. This concept is of general value and it is likely that any framework can benefit from a framelet-based design approach just as the AOCS framework did.

In a second contribution of this paper, section 4 argues that the AOCS framework can be seen as an extension of the operating system. This conceptualisation of frameworks will be possible whenever the target applications can be broken up into distinct and independent functionalities for in that case the manager design meta-pattern will allow functionality managers to be defined that behave like OS extensions.

Section 6 discusses the constraints imposed on the framework design by the real-time nature of the AOCS framework. It is unlikely that a general solution to the real-time problem in framework design can ever be found. Probably, the most a designer can hope for is a set of guidelines that, having been found useful in a particular context, might be reused again, perhaps with suitable modifications. The solutions described in section 6 should be

seen in this light: as additions to the store of experience on frameworks for real-time systems.

## REFERENCES

[1] M. Fayad, D. Schmidt, R. Johnson, *Application Frameworks*, p. 3-28, M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks*, Wiley Computing Publishing, 1999

[2] G. Pomberger, W. Pree, *Quantitative and Qualitative Aspects of Object-Oriented Software Development*, International Symposium on Object-Oriented Methodologies and Systems (ISOOMS '94, Springer-Verlag), Palermo, 21-23 September 1994

[3] W. Pree, K. Koskimies, *Framelets – Small is Beautiful*, p. 411-414, M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks*, Wiley Computing Publishing, 1999

[4] M. Mattsson, J. Bosch, *Composition Problems, Causes, and Solutions*, p.467-487, M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks*, Wiley Computing Publishing, 1999

[5] M. Awad, J. Kuusela, J. Ziegler, *Object Oriented Technology for Real Time Systems*, Prentice Hall, 1996

[6] W. Pree, *Meta Patterns – A Means of Capturing the Essential of Reusable Object Oriented Design*, Proceedings of the 8th European Conference on Object-Oriented Programming, Bologna, Italy, July 1994

[7] C. Szyperski, (1998), *Component Software—Beyond Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley

[8] Society of Automotive Engineers, *Generic Open Architecture (GOA) Framework*, SAE Document AS4893, Jan. 1996

[9] E. Gamma *et al.*, *Design Patterns – Elements of Reusable Object Oriented Software*, Reading, Massachusetts: Addison-Wesley, 1995

[10] J. Wertz (ed), *Spacecraft Attitude Determination and Control*, Kluwer Academic Publisher, 1995

[11] W. Larson, J. Wertz (eds.), *Space Mission Analysis and* Design, Kluwer Academic Publishers, 1997

[12] A. Pasetti, W. Pree, *Two Novel Concepts for Systematic Product Line Development*, in P. Donohoe (ed), *Software Product Lines*, Kluwer Academic Publisher, 2000

[13] W. Pree, *Hot-Spot Driven Development*, p. 379-394, M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks*, Wiley Computing Publishing, 1999

[14] A. Burns, A. Wellings, *HRT-HOOD:A Structured Design Method for hard Real Time Systems*, Real-Time SYstems, 6, p. 73-114, 1994

[15] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, Addison Wesley Longman, 1998

[16] R. Johnson, *Frameworks=(Components+Patterns)*, Communications of the ACM, Vol. 40, N. 10, p. 39-42, Oct. 1997

[17] W Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995

[18] A. Weinand, E. Gamma, R. Marty, *ET++ - An Object-Oriented Application Framework in C++*, OOPSLA'88, Special Issue of SIGPLAN Notices, Vol. 23, No. 11, 1988

[19] Taligent, Inc., *The Power of Frameworks*, Reading, Massachusetts: Addison-Wesley, 1995

[20] K. Pirklbauer, R. Plösch, R. Weinreich, *Object-Oriented Process Control Software*, Journal of Object-Oriented Programming, April 1994
[21] A. Goldberg, *Smalltalk-80 / The Interactive Programming Environment*, Addison-Wesley, 1984
[22] W. Pree, K. Koskimies, *Rearchitecturing Legacy Systems—Concepts & Case Study*, WICSA '99: First Working IFIP Conference on Software Architecture, San Antonio, Texas, 22-24 Feb. 1999
[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J Loingtier, J. Irwin, *Aspect-Oriented Programming*, European Conference on Object-Oriented Programming (ECOOP `97, Springer-Verlag), Finland, June 1997
[24] Aspect-Oriented Programming homepage at Xerox PARC, http://www.parc.xerox.com/csl/projects/aop