THE AOCS FRAMEWORK

V. Cechticky[*], G. Montalto[**], A. Pasetti [*], N. Salerno[**]

[*]*Automatic Control Laboratory, ETH-Zürich, CH-8092 Zürich, Switzerland*
[**] *Alenia Spazio S.p.A., via Saccomuro 24, 00131 Rome, Italy*
cechti@aut.ee.ethz.ch, pasetti@pnp-software.com, g.montalto@roma.alespazio.it, n.salerno@roma.alespazio.it

This paper presents the *AOCS Framework.* 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 AOCS Framework applies this technology to the Attitude and Orbit Control Subsystem (AOCS) domain. Its aim is to allow the rapid instantiation of the AOCS software and of other on-board control systems. The AOCS Framework is being used at Alenia Spazio to develop the software for the antenna pointing control (APC) subsystem of geo-stationary platforms. The development of the AOCS Framework was partly funded under Estec contract 13776/99/NL/MV.

The first part of the paper presents framework technology and shows its relevance to the AOCS domain. The second part presents the AOCS Framework. The third part presents the experience in the application of the AOCS Framework to the APC domain. The fourth part presents future directions of development.

1. SOFTWARE FRAMEWORKS AND THE AOCS

It is a common situation that applications within the same domain tend to share the same type of high-level requirements. This commonality of requirements often is reflected in a commonality of structure at software level. Thus, for instance, in the AOCS domain one will normally find requirements defining processing of sensor measurements. Although the particular algorithms that are used to perform this processing vary from application to application, the general structure of the processing remains the same. Examples of the invariant features of this processing in the AOCS domain are:

- the sensor measurements are acquired and must be processed on a cyclical basis,
- the processing algorithms are mode-dependent,
- checks must be performed on the size of certain variables (e.g. control error, compensator output, etc) to detect possible failures,
- recovery actions must be taken in response to detected failures,
- the results of key operations (e.g. computation of control torques) should be available for downloading in telemetry,
- it should be possible for key parameters in the processing algorithms (e.g. the gains in a PID controller) to be tuned by telecommand.

This type of *structural similarity* is seldom modeled in software applications. Software reuse, to the extent that it takes place at all, is usually restricted to code fragments and even in that case it is only possible if the same person or the same team have been involved in different projects. This situation is certainly typical of AOCS systems where low level of software reuse is one of the causes of escalating costs.

Software frameworks [Gam95, Fay99] have been introduced to address this problem of reuse in the large. More specifically, a software framework proposes an architecture that is optimized for all applications within a certain domain and is designed to allow rapid instantiation of applications within that domain. Experienced software engineers who develop several related applications reuse architectures as a matter of course. Software frameworks 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 and a software framework can be the basis of a domain-specific autocoding tool.

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. 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.

The AOCS framework would have to model all the invariant features of this domain (some of which were listed above). Additionally, it would have to offer mechanisms through which the invariant architecture can be tailored to match the specific requirements of a particular AOCS application. The figure below shows a framework for an AOCS. The framework defines the architectural backbone of the application (shaded in the Figure 1). This is invariant in the application's domain and is provided by the framework. This invariant structure is customized by combining it with software components that model the application-specific part of the application (unshaded block in the Figure 1). The Figure 1shows two such customization components.

If a framework approach is adopted, the development of an application software takes the form of a process of customization of the framework itself. The advantage of this approach is that customization of an existing artifact is much easier to perform than development from scratch. Note that a framework approach has a profound impact on the software development process. The definition of the requirements of a new application cannot take place in a vacuum but must be made with reference to the functionalities offered by the framework. This introduces some constraints on the specification of the application but can also make the specification process faster and more precise.

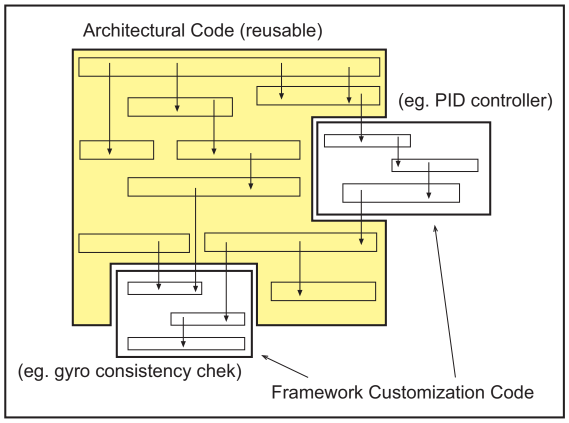
Architectural Code (reusable)
(eg. PID controller)
(eg. gyro consistency chek)
Framework Customization Code

Figure 1. The framework architecture

A further advantage is that use of a framework helps bridge the gap between the application specialist and the software specialist. Consider again the case of the AOCS domain. Normally, the AOCS engineer produces a set of user requirements that are then handed over to a software specialist for implementation. The latter however is normally *not* an AOCS specialist and the interface between the two disciplines – the AOCS discipline and the software discipline – is often a source of misunderstandings, schedule delays and cost overruns. Software frameworks promote a more interdisciplinary approach where the job of the software specialist is less to develop specific applications on behalf of the application specialists than to develop tools that empower the application specialists to directly build their own applications. In this sense too a framework approach is very similar to an autocoding approach.

2. THE AOCS FRAMEWORK – DESIGN AND DEVELOPMENT

The design of a framework for the AOCS began by identifying the typical functionalities implemented by an AOCS [Pas02d]. The type of AOCS that was taken as reference is the kind that is normally used in ESA science missions (e.g. ISO, SOHO, XMM) where the AOCS subsystem is controlled by a dedicated processor. The functionalities that are modeled by the AOCS Framework accordingly are:

- *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)
- *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)

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. A 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 be described for the case of the telemetry and controller functionalities.

2.1 The AOCS Framework Structure

The structure of the AOCS assumed by the AOCS Framework is schematically shown in the Figure 2.

Note that in some cases (for instance, in many ESA earth-observation missions) the AOCS software shares the same processor as the OBDH software. In such cases, the AOCS Framework can only be used in part because some of its functionalities may already be provided by the OBDH software.

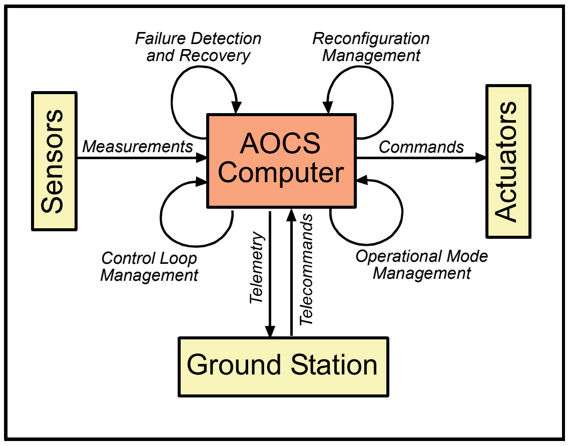

Figure 2. The AOCS framework structure

2.2 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.

The approach applied in the AOCS framework for the telemetry functionality 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:

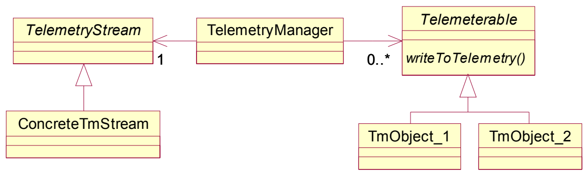

Figure 3. The telemetry design pattern

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:

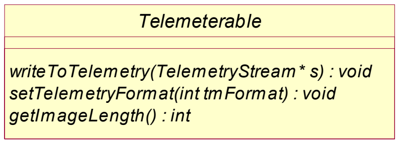
Telemeterable
writeToTelemetry(TelemetryStream s) : void*
setTelemetryFormat(int tmFormat) : void
getImageLength() : int

Figure 4. The Telemeterable interface

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).

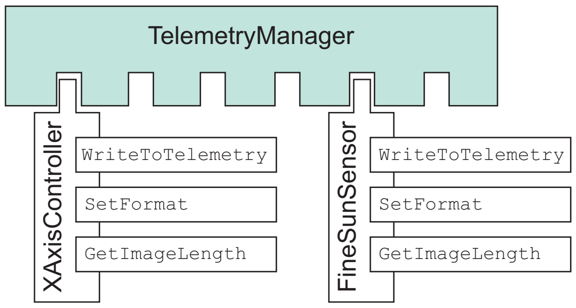

Figure 5. Plug-in view of a telemetry manager

The Figure 5 shows a plug-in view of telemetry management. The reusability originates in the introduction of an abstract interface to separate the management of functionality from its implementation.

2.3 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 *controller design pattern* is introduced for the purpose to separate the management of controllers from the implementation of the (application-specific) control algorithms:

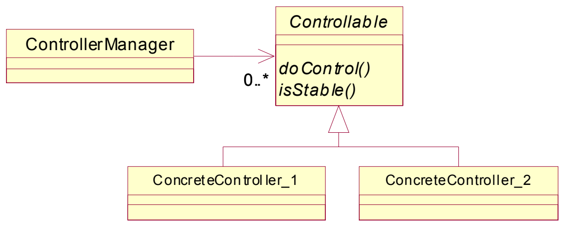

Figure 6. The controller design pattern

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 the 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 the telemetry manager with a controller manager that is customized with plug-in components of `Controller` type.

2.4 The Manager Meta-Pattern

The telemetry and controller design patterns are very similar. 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. 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:

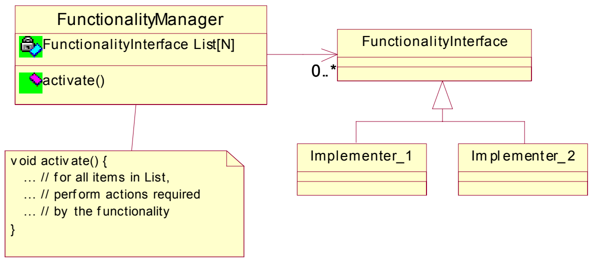
FunctionalityManager
FunctionalityInterface List[N]
activate()
FunctionalityInterface
0..*
Implementer_1
Implementer_2
void activate() {
... // for all items in List,
... // perform actions required
... // by the functionality
}

Figure 7. The manager meta-pattern

The manager meta-pattern was applied to all the AOCS functionalities. 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.

3. THE AOCS FRAMEWORK - APPLICATION

The prototype AOCS framework was initially tested by using it to develop a simplified AOCS software for the ERC32 processor running the RTEMS operating system.

A more ambitious application of the framework concept is currently under way at Alenia-Spazio [Mon02]. One of the domains of interest for Alenia Spazio is the antenna pointing control (APC) subsystem on board geo-stationary platforms. An APC subsystem (see figure below) is an on-board control system and its structure is very similar to that of the AOCS. The Tracking Receivers (TRRs) sense the position of the antenna and the Antenna Pointing Drivers (APDs) act as actuators. The APC software processes the TRR signals and generates signals for the APDs that control the orientation of the antennas.

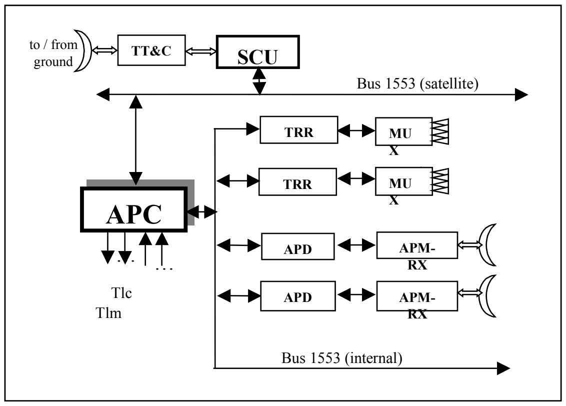

Figure 8. The antenna pointing control (APC)

The structural similarity to the AOCS means that the AOCS Framework is applicable, at least in part, to the APC domain. This domain is also well-suited to validate the framework concept because it is less mission-critical than the AOCS.

The adoption of a framework approach for the APC domain is part of a wider drive by Alenia Spazio to improve software reuse. Object-oriented frameworks are seen as a key technology because of their ability to support architectural-level reuse.

4. CONCLUSIONS AND FUTURE DEVELOPMENTS

Experience to date with the AOCS Framework has been very positive. The results obtained at Alenia Spazio in particular are encouraging and confirm that framework technology has much to offer to on-board control systems. Future work will follow two directions. On the one hand, concrete applications of the frameworks in the AOCS field will be sought. On the other hand, attention will be devoted to automatizing the framework instantiation process. The analogy between frameworks and autocoding tools has already been noted. This analogy can be pushed further by attaching to the framework an environment where the instantiation process can be assisted by autocoding techniques. Development of such an environment is being pursued in an ESA study that starts in June 2002 under contract AO/1-3959/01/NL/PB [Pas02c].

5. REFERENCES

[Fay99] Fayad M, Schmidt D, R. Johnson R (eds) *Building Application Frameworks – Object Oriented Foundations of Framework Design*. Wiley Computer Publishing, 1995

[Gam95] E. Gamma et. al., *Design Patterns*, Addison Wesley, 1995

[Mon02] G. Montalto, A. Pasetti, N. Salerno, *Application of Software Framework Technology to an Antenna Pointing Controller*, DASIA 2002, Dublin, May 2002

[Pas02a] http://www.aut.ee.ethz.ch/~pasetti/AocsFramework/index.html

[Pas02b] http://www.aut.ee.ethz.ch/~pasetti/RealTimeJavaFramework/index.html

[Pas02c] http://www.aut.ee.ethz.ch/~pasetti/JavaBeansFramework/index.html

[Pas02d] A. Pasetti, *Software Frameworks and Embedded Control Systems*, LNCS Vol. 2231, Springer-Verlag, 2002