A REUSABLE ARCHITECTURE FOR SATELLITE CONTROL SOFTWARE

A. Pasetti and W. Pree, Dept. of Computer Science, Univ. of Constance, D-78457, Constance, (Germany)

Introduction

It is now recognized that software reuse, to be truly effective, needs to go beyond the reuse of mere code fragments or modules. The toughest design choices, and those where errors and non-compliances most often arise, concern the architecture of a software system. It is accordingly this software architecture that must be made reusable in order to make the intellectual investment that went into developing it available to multiple projects. In mainstream applications, this form of reuse has led to the construction of *software frameworks* [1,2,3], namely artifacts that encapsulate an architecture optimized for all applications within a certain domain and that provide a ready-made skeleton from which individual applications within that domain can be instantiated.

This approach has brought unquestioned benefits to fields as diverse as GUI, business and system applications but has so far been shunned in the real-time and embedded world in general and in space applications in particular. Quite apart from the conservatism of space software designers – a consequence of the high reliability requirements of their applications – software frameworks have been avoided in this field because of the processing and memory overheads they introduce and because, often, they are based on object-oriented (OO) languages that are felt to be inappropriate for mission-critical applications. Space qualification of processors like the ESA's ERC32 (built around a 32-bit SPARC architecture [4]), the introduction of "safe" OO languages like Ada95 and the emergence of guidelines for the use of C++ in mission critical systems (see for instance [5]) in the view of the authors make such concerns obsolete.

This paper is based on a project funded by the European Space Agency[1] to design and prototype an object-oriented software framework for the Attitude and Orbit Control Subsystem (AOCS) of satellites. At present, the prototype framework is being readied for deployment on an ERC32 processor. This paper outlines its architecture.

The AOCS

Figure 1 shows the structure of an AOCS. The AOCS is an embedded hard real-time control system. Its chief task is to periodically collect measurements from sensors and convert them into commands for 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 should be able to survive some types of faults and prolonged periods of ground station outage. Robustness to faults and autonomy require the AOCS to perform failure detection and failure recovery actions.

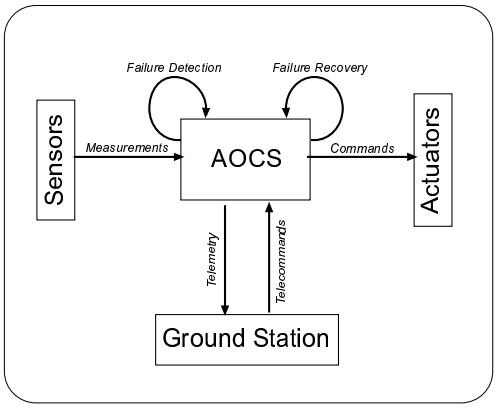


Figure 1 : Structure of AOCS

[1] 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.

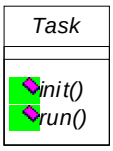
The AOCS software is usually organized as a set of statically defined tasks running under the control of a cyclical scheduler. Its code size normally lies between 10000 and 20000 lines of very compact code. This will increase in the future as more powerful space-qualified processors come into use. The software is normally written in Ada83 or C with sprinklings of assembler.

The RTOS Reuse Model

The objective of the AOCS framework project is to improve software reuse in a particular kind of real-time, embedded applications. Real Time Operating Systems (RTOS) constitute a very successful – and often overlooked – example of software reuse in the real-time field. For over two decades, packages have been available as commercial off-the-shelf products that offer generic solutions for the management of most functionalities related to task scheduling. The reuse model behind RTOS is based on two key features.

There is firstly the *enforcement of an architectural infrastructure*. RTOS's assume an application to be made up of tasks with certain well-defined characteristics (single entry point, mutual exclusion mechanisms, etc). Applications that wish to use the RTOS must conform to this architecture.

Secondly, the RTOS relies on the *separation of the management of a functionality from its implementation*. Consider for instance task scheduling. At its simplest, and using UML notation, the RTOS sees a task as an object derived from the following abstract class:



where a call to `init()` causes the task to initialize itself and a call to `run()` causes it to start executing. Separation through an abstract interface is essential since the *implementation* of the task is obviously application-dependent and hence reusability can only concern the *management* of the scheduling functionality.

The reuse approach taken in the AOCS framework follows the RTOS model in enforcing an architectural infrastructure and in splitting each the AOCS functionalities in functionality management components – that are made reusable – and in functionality implementers that are realized as application-dependent components to be plugged into the framework.

The next section presents the architectural infrastructure that the framework enforces and the following section presents some of the functionality managers. More detailed information can be obtained from [15].

Architectural Infrastructure

The architectural infrastructure is built around design patterns [6, 7] that address design problems specific to the AOCS domain. The main such problems and the proposed solutions are outlined in the subsections below.

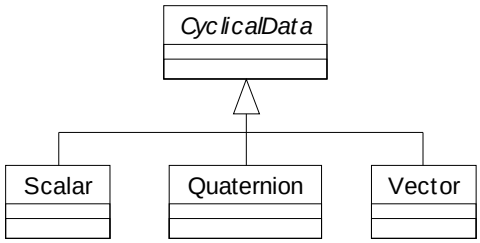
Inter-component Communication

The AOCS framework builds the AOCS software as a collection of independent and cooperating components. The inter-component communication mechanism is loosely based on the JavaSpace model [8].

Two categories of data are recognized by the framework: *cyclical data* and *event data*. The former are data produced or consumed on a periodic basis by AOCS components (eg sensor outputs). The latter are data produced asynchronously by components that wish to signal a change in their internal state or the occurrence of some event such as the execution of a telecommand or the detection of a failure.

Both cyclical and event data can be of different kinds (eg cyclical data can be quaternions, vectors, scalars, etc) but both are derived from a single abstract class. Thus, for instance, for cyclical data, a class structure of the following kind results[2]:

[2] All UML class diagrams shown in this paper are highly simplified showing only the main operations of each class.



Housekeeping operations are performed through the abstract base class so that data are treated in a uniform manner independent of their specific type.

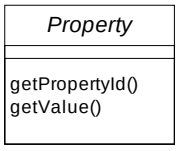
Cyclical and event data reside in shared memory areas called *data pools* for the cyclical data and *event repositories* for the event data. Access to shared data is through accessor methods so that this solution is compatible with a distributed architecture where the AOCS software is spread over several processors or processes.

Each item in a data pool has a owner component which is the only one authorized to change its value. Other components have unrestricted read access. Events in a repository can be inspected but not copied out of the repository. This ensure that only authorized components process them.

Property Monitoring

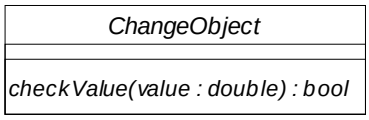
The property monitoring mechanism was introduced primarily to allow AOCS components to coordinate their behaviour.

A *property* is an attribute of a component that describes one aspect of its behaviour or of its internal state that is made accessible to external components. Properties are encapsulated in objects derived from a common abstract class `Property`:



Components coordinate their behaviour by monitoring each other's properties. The property model proposed by the AOCS framework is derived from the JavaBeans architecture [9]. The main addition is the introduction of *change objects*.

The term monitoring refers to the observation of a change over time in the value of a property. At the most basic level, monitors are interested in *any* change in the monitored property. More frequently, however, they are only interested in *certain types* of changes. In AOCS systems, monitors are typically interested in variables exiting a pre-defined range or in their changing by more than a pre-defined delta-threshold. In order to avoid burdening monitors with the need to implement highly specific (and hence non-reusable) filters, the concept of change is encapsulated in objects derived from an abstract class called `ChangeObject`:



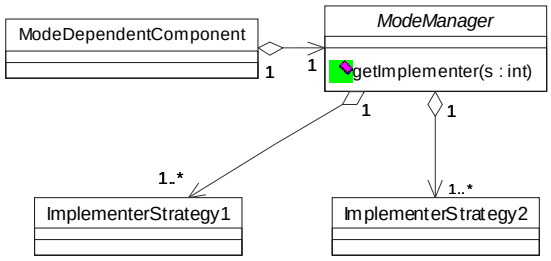
Thus, a check on a property is made by passing its value to method `checkValue` that returns `true` if the change encapsulated in the change object has occurred.

The framework offers three property monitoring mechanisms ranging from a simple direct check on a property's value to a registration mechanism whereby the monitor registers with the monitored component asking to be notified if a certain kind of change (as specified by a change object) occurs. The use of property and change objects derived from abstract base classes allows the monitoring mechanisms to be implemented in a generic manner independent of both the property being monitored and the type of change being checked.

Operational Mode

Current AOCS systems are based on the concept of operational mode. The operational mode is an attribute of the AOCS software as a whole. Its purpose is to adapt the software's behaviour to various sets of external conditions. In keeping with a component-oriented approach, the AOCS framework instead makes operational mode an attribute of individual components.

Mode-dependent components have the structure shown below:

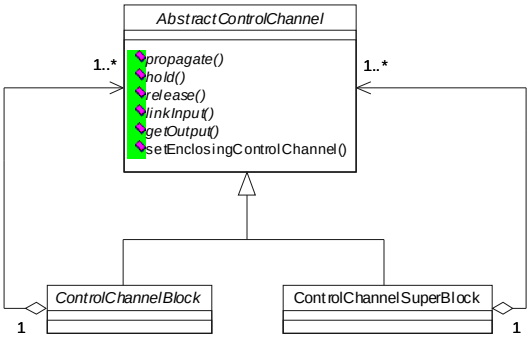


The mode-dependent component relies on one or more strategies. Each strategy has several implementations. To each operational mode, there corresponds a set of strategy implementations. At any instant in time, the mode-dependent component retrieves the strategy implementers adequate to current operational conditions from its mode manager using method `getImplementer(s)` where s is the strategy number. The mode manager uses the property monitoring mechanisms outlined above to keep track of changes in its environment and to decide when to perform mode switches. An attitude controller, for instance, relies on a mode manager to supply it with the control algorithm corresponding to the current operational mode.

This architecture – based on an extension of the strategy pattern of [6] – separates the implementation of mode-specific algorithms from the implementation of mode-switching logic. It is described in more detail in [16].

Sequential Data Processing

Cyclical data in an AOCS go through several processing stages. In the AOCS framework, processing of such data flow data is done in *blocks*. Blocks can be chained together and block chains can be nested within *superblocks*. The data structures representing these abstractions are:

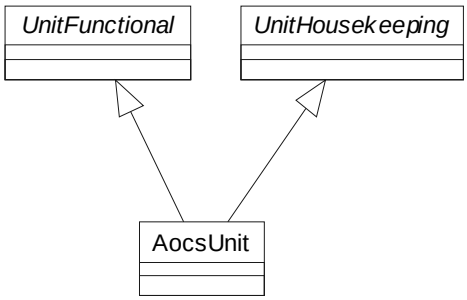


Concrete blocks are instances of subclasses of `ControlChannelBlock`. Superblocks are instances of `ControlChannelSuperBlock`. Base class `AbstractControlChannel` encapsulates generic operations that can be performed on all data processing structures, regardless of whether they are blocks or superblocks. It thus allows blocks and superblocks to be mixed when constructing concrete processing structures.

The AOCS framework offer pre-defined blocks implementing a PID filter and embedding code generated from the Xmath autocode tool.

External Unit Management

External units (sensors and actuators) are represented within the AOCS framework by *proxy objects* implementing two interfaces:



Interface `UnitFunctional` defines operations needed for data exchanges with the physical unit. It captures the behaviour of AOCS units that is visible to the unit's clients. Interface `UnitHousekeeping` defines housekeeping operations such as unit switch-on and -off, unit self-test, etc. Class `AocsUnit` was split into two interfaces to introduce the concept of *fictitious unit*. Data processing components (eg controllers) do not always interface directly with units. Often, pre-processing components (unit reconfiguration managers, filters, unit selectors, etc) are interposed as in figure 2. The preprocessing components are called fictitious units because they are made to implement the `UnitFunctional` interface and therefore "look like" units. This makes it possible to freely interchange preprocessing components and it insulates the data processing components from the type of preprocessing that is performed on unit data.

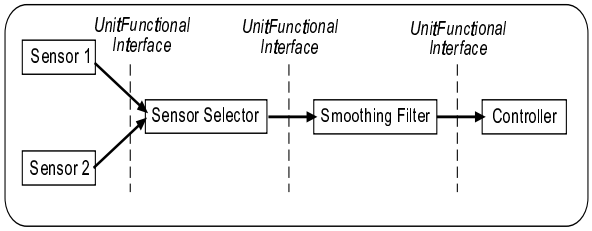


Figure 2: Sensor Processing Chain

Units are too diverse to be packaged in standard configurable components. Reusability is fostered by making their software proxies conform to a standard interface thus decoupling the managers and user of unit data from the unit themselves.

Functionality Managers

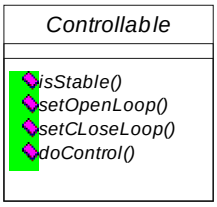
The AOCS framework provides a number of functionality managers as generic components that can be deployed and reused "as is" in any AOCS.

The functionality managers interact with the (application-dependent) components they manage through abstract interfaces that define the operations that they can perform upon them. The implementation of these operations is application-specific and has to be defined for each particular AOCS. Functionality managers are active components that are periodically activated by a scheduler (not part of the framework).

The following subsections describe the main functionality managers offered by the framework.

The Controller Manager

An AOCS typically contains several closed-loop controllers to control the spacecraft attitude on each axis, the speed of the reaction wheels and, perhaps, the spacecraft orbit. In the AOCS framework, controllers are encapsulated in objects that implement the following interface:



This interface defines the operations that make sense on controller. The main such operations are:

`isStable`() that returns true if the controller is stable; `setCloseLoop`() and `setOpenLoop`() to set the control loop to closed/open; and `doControl`() to compute and apply the control action.

The operation of the controller manager can be described in terms of such operations. In simplified terms, the actions taken by the controller manager when it is activated are:

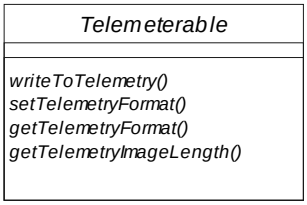
```
Controllable* list[N];
. . .
for (all objects 'c' in list) do
{  if (!c.isStable())
       . . .  // instability!
   else
      doControl();
 }
```

Thus, the controller manager maintains a list of controllers – which are seen as instances of abstract class `Controller` – and it periodically asks each controller to verify its own stability and, if stability is confirmed, it asks it to compute and apply its control action. The controller manager moreover exposes operations allowing items to be dynamically added to or removed from the list, individual controllers to be put in open or closed loop mode, and other generic operations to be performed on the controllers it manages. The list of controllers finally is mode-dependent which means that it is supplied at each activation of the controller manager by a mode manager component.

The Telemetry Manager

In current AOCS's, telemetry handlers directly collect telemetry data storing them in buffers from which they are transferred to the ground. They thus need an intimate knowledge of the type and format of the telemetry data. This coupling between handler and data makes the former mission-specific and hinders its re-use.

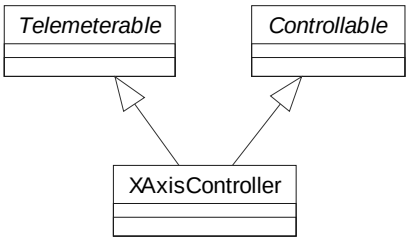
In the AOCS framework the software is organized as a collection of components each potentially capable of *writing itself* to telemetry. This means that each component implements the following abstract interface:



The basic method is `writeToTelemetry`. Calls to it cause the object to write its internal state to the telemetry stream. The telemetry manager simply keeps track of the telemetry objects and periodically calls their `writeToTelemetry` method. It uses the other operations exposed by `Telemeterable` to check the size and format of the telemetry image.

As usual, operations to dynamically manage the items in the list are provided and the list itself is mode-dependent being supplied by a mode manager. More details on the telemetry manager can be found in [16].

Note that the same object may be managed by several functionality managers at the same time. Thus, for instance, the component implementing the X-axis attitude controller will in general be managed by both the controller manager and the telemetry manager. In practice, this means that this component implements two interfaces:

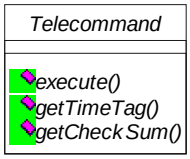


In fact, most framework objects typically implement 6-7 different interfaces since they interact with several functionality managers. Multiple inheritance of interfaces (not of implementation!) therefore plays a key role in the AOCS framework.

The Telecommand Manager

Traditionally, telecommands are strings of data bytes beginning with a header word identifying the telecommand type followed by data words and terminated by a checksum. Such telecommands are executed on-board by a telecommands handler essentially consisting of a `case` statement that processes each telecommand according to its type as defined by the telecommand identifier.

The AOCS framework breaks with this concept. It applies the command design pattern from [6] and treats telecommands as objects implementing the following abstract interface:

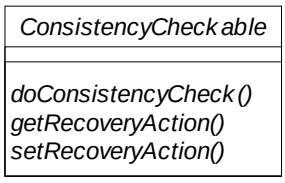


The key method is `execute` which, when called, performs the actions associated with the telecommand itself. The telecommand manager therefore maintains a list of objects of type `Telecommand` and cyclically goes through the list, verifies the time tag and check-sum of each telecommand and, if appropriate, asks it to execute itself.

Note that with this concept the code associated to a telecommand may be part of the telecommand itself and may have to be uplinked to the spacecraft alongside the telecommand data. More details on the telemetry manager can be found in [16].

The Failure Detection Manager

The most typical failure detection test performed on an AOCS is a *consistency check* where a failure is declared if an object is found to be in an inconsistent state (eg. the sun sensor and the gyro read-outs indicate different angular velocities). This type of failure detection test is covered by the AOCS framework through the introduction of the following abstract interface:



Objects that may be potentially subjected to consistency checks are made to implement interface `ConsistencyCheckable`. A call to method `doConsistencyCheck` causes the consistency check to be executed. The method returns `false` if the check failed. The getter and setter methods refer

to the recovery action associated to the failure (see below). The failure detection manager maintains a list of objects of type `ConsistencyCheckable` and periodically asks them to perform a consistency check upon themselves.

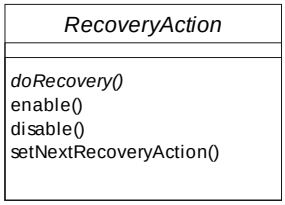
Additionally, failure detection is performed by comparing the values of key parameters – such as the output of controllers and sensors – against pre-defined limits or by verifying that they do not undergo sharp changes. The failure detection manager encapsulates all such parameter values in *property objects* and the limits or type of change against which monitoring must be performed in *change objects*. The monitoring action can then be done systematically and independently of the particular parameters being checked.

The type of failure checks to be done depends on operational conditions. This is modeled by making the list of `ConsistencyCheckable` objects and of properties to be monitored mode-dependent: both list are supplied by a dedicated mode manager component.

The Failure Recovery Manager

The AOCS framework regards failure detection and failure recovery as separate tasks allocated to two distinct active components. The failure detection manager performs failure checks and, when it detects a failure, it creates an event that describes it and stores it in an event repository. At a later time, the failure recovery manager inspects the failure event repository to check whether any failures have been identified and then uses the information stored in the failure events to decide on the appropriate response. The chief advantage of this approach is that a separate failure recovery manager can inspect *all* failures detected in a certain control cycle and can implement recovery responses that take account of the interrelationships of different individual failures.

Failure recovery is performed at two levels: at the level of *recovery action* and of *recovery strategy*. A recovery action encapsulates a single action taken in response to a single failure event. Recovery actions are encapsulated in objects that implement the following interface:

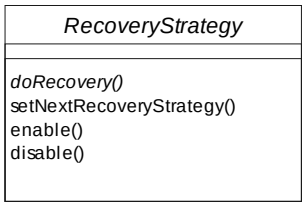


The key method is `doRecovery` that causes the recovery action to be executed. Method `setNextRecoveryAction` allows several recovery actions to be chained together so that a call to `doRecovery` causes all actions in the chain to be executed in sequence.

The AOCS framework stipulates that to each failure a recovery action must be associated: components that perform a failure check, should know what has to be done if the check fails and this knowledge is contained in a recovery action object. In particular, components implementing interface `ConsistencyCheckable` must be able to supply a recovery action object specifying the response to the failure of the consistency check.

Typical recovery actions are: resetting a component, resetting the AOCS software, performing a unit reconfiguration. The framework offers predefined components encapsulating these and other common recovery actions.

While recovery actions represent responses to individual failure events, a recovery strategy represents a set of coordinated responses to all the failure events in the failure event repository. As usual, recovery actions are encapsulated in objects derived from an abstract interface:

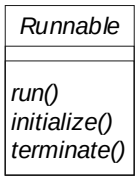


A call to `doRecovery` causes the strategy to be executed. Like recovery actions, recovery strategies can be linked together. The recovery manager is responsible for executing recovery strategies. The chain of responsibility pattern from [6] is used to pass execution orders along chains of linked recovery strategies.

A simple recovery strategy could be to check the number of failure events generated in the last period and, if this larger than a certain threshold, to reset the AOCS software. A more complex recovery strategy might instead execute in sequence all the recovery actions associated to the failure events currently in the failure event repository. The framework offers ready-made components implementing both of these default strategies.

Scheduling

AOCS's are multi-task systems. The AOCS framework distinguishes active objects that implement the following interface:



where `run` is a task's entry point. The framework does not enforce any specific scheduling policy and does not include any mutual exclusion mechanisms which, if pre-emptive scheduling is adopted (against current practice in the AOCS field) would have to be built on top of the framework components.

The framework design ensures that AOCS applications derived from the framework are HRT-HOOD compatible in the sense that their schedulability properties can be established by static analysis [10].

Design Methodology

Design methodologies currently in use for AOCS – such as HOOD [11] – tend to be object-based and are therefore unsuitable for the systems like the AOCS framework that are truly object-oriented in the sense of relying heavily on abstract coupling and dynamic typing. There is at present no well-established design methodology for software frameworks. The AOCS framework project was used as a test case to propose a methodology of this kind and the results are reported in [12].

Implementation Issues

The AOCS framework is implemented in C++ with the GNU compiler. Initially, Ada95 was favoured but was finally discarded because of its poor support for multiple inheritance. C++ is not as "safe" a language as Ada but in a component-based system language issues become less important. Language choice affects the "inside" of a component but, when a system is built by assembling pre-defined components, one assumes that the components are error-free (they have been tested by their supplier). Attention therefore shifts to the component configuration and assembly process that occurs at a level higher than that of ordinary programming languages.

Because of the real-time environment, both dynamic memory allocation and exceptions are avoided. All non-trivial objects are created statically and never destroyed. Such objects are derived from a common base class (similar to Java's `Object` class) whose destructor is declared private so that inadvertent dynamic destruction is caught by the compiler.

The framework design relies heavily on multiple inheritance which is used in the safe form proposed by the Java language, namely it is allowed only from pure virtual classes. Implementation inheritance is only allowed from a single base class. This removes the well-known problems associated to multiple inheritance [13].

Estimating the memory footprint of a framework is difficult because not all components offered by a framework go into all applications instantiated from the framework. Preliminary tests indicate that, on the average, a framework class occupies 2 kbyte (memory+data). A typical AOCS might include between 100 and 150 such classes thus giving a footprint of 200 to 300 kbytes. This fits easily into the multi-megabytes memory of the ERC32.

The AOCS framework is currently in the last stage of its development. Many of its components have already been deployed on an ERC32 simulator [4] where final testing will be done in the second half of this year. Time and schedule permitting, tests will also be done on an ERC32 test board.

Conclusions

The software development process in the AOCS field –indeed in the space field in general – typically begins with the definition of user requirements. Requirements are normally formulated under the tacit assumption that all the software will be developed anew. Consequently, they are narrowly targeted at a specific mission and result in a monolithic piece of code that, while highly optimized for the application at hand, has to be developed entirely from scratch and cannot be reused in other missions.

This approach is in sharp contrast to that adopted on the hardware side. Here, designers begin by surveying the market for available components and then specify their system in terms of these components. Widely accepted standards (on bus interfaces, on electrical interfaces, on mechanical interfaces, etc.) allow components from different suppliers to be plugged together. The resulting system is perhaps not optimal in terms of mass or power consumption (the characteristics of standard components seldom match perfectly the requirements of a specific project) but it is certainly cheaper and faster to assemble than if it were developed from scratch.

The research group at the University of Constance to which the authors belong, has traditionally been concerned with software architectures for complex systems in the business field where it has advocated and successfully applied the component and framework approach which tries to build software very much like hardware is built: by assembling pre-defined components with standardized interfaces. The AOCS framework project is part of an effort to port this technology to a real-time, mission-critical field (see [14] for the long-term goals of this effort). While a final assessment will have to wait for the complete deployment of the AOCS framework (foreseen for the end of 2000), results to date indicate that the introduction of advanced processors in space applications makes framework and component technologies viable options for space systems promising to bring them the same benefits in terms of reusability and reduced development times seen in other fields.

References

[1] Fayad M, Schmidt, Johnson (Ed.), 1999, *Building Application Frameworks*, Wiley Computing Publishing

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

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

[4] www.estec.esa.nl/wsmwww/erc32/erc32.html

[5] http://hissa.ncsl.nist.gov/sw_develop/safety.html

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

[7] Pree W. (1995), *Design Patterns for Object-Oriented Software Development*, Addison-Wesley

[8] Freeman E. (1999), *JavaSpaces Principles, Patterns, and Practice*, Addison-Wesley

[9] Englander R. (1997), *Developing Java Beans*, O'Reilly & Associates Inc.

[10] Burns A, Wellings, 1994, *HRT-HOOD: A structured Design Method for Hard Real-Time Systems*, Real-Time Systems, Vol. 6, pag. 73-114

[11] http://www.estec.esa.nl/wmwww/WME/oot/hood/index.html

[12] Pasetti A, Pree, 2000, *Two Novel Concepts for Systematic Product Line Development*, to be published in the Proceedings of the First Software Product Line Conference; 28-39 Aug 2000, Denver, Colorado (USA)

[13] Szyperski C., 1998, *Component Software*, Harrow (UK), Addison Wesley Longman Limited

[14] Pasetti A, Pree, 1999, *The Component Software Challenge for Real-Time Systems*, Proceedings of the 1-st International Workshop on Real-Time Mission-Critical Systems; 30 Nov - 1 Dec 1999, Scottsdale, AZ (USA)

[15] http://www.softwareresearch.net/AocsFrameworkProject/ProjectHomePage.html

[16] Pasetti A, Pree, 1999, *A Component Framework for Satellite On-Board Software*, Proceedings of the 18-th Digital Avionics System Conference; Oct. 1999, St. Louis, Missouri (USA)