A COMPONENT FRAMEWORK FOR SATELLITE ON-BOARD SOFTWARE

A. Pasetti, W. Pree,

C. Doppler Lab. for Software Research, Univ. of Constance, Constance, Germany

Abstract

This paper advocates a new approach to satellite software design based on object-oriented *framework technology* and describes early results from a project for the European Space Agency (ESA) to design a software framework for satellite attitude and orbit control systems[1] (AOCS).

Frameworks are collections of components with pre-defined cooperations among them. They make *architecture* (as opposed to mere *code*) reuse possible. The framework concept is being tested in a re-design of the AOCS software. This paper illustrates it by describing the implementation of telecommand handling, telemetry handling, and operational mode management.

The Problem

The experience of one of the authors on several major European space projects is that the software for an AOCS system tends to be developed from scratch for each new mission. Failure to reuse code, in space as in other fields, is a well-known problem firmly rooted in economic facts. Figure 1 summarizes the results of a study[2] regarding the reuse costs of single components in NASA projects. The graph relates the percentage of necessary changes to a single component in order to render it reusable (x-axis) to the costs of these changes relative to the development of a component from scratch (y-axis).

The figure shows that even small changes (12%) raise the reuse costs to a significant share (55%) of the rewrite costs. Moreover, even reusing components without any changes does not come for free: catalogs of reusable components have to be maintained and building components for reuse is more expensive than building them for a special purpose.

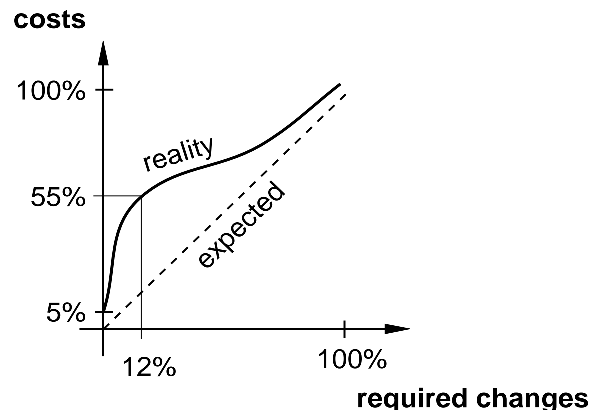

Figure 1: Relative Cost of Software Reuse

A more serious limitation of reuse is that traditionally its scope has been limited to code fragments. With the conventional paradigm - prevalent in space projects - that separates functions and data, software reuse means at best the reuse of individual subroutines or modules. The *software architecture* - often the toughest part to design - cannot be ported at all. Experienced designers will reuse architectures as a matter of course but, traditionally, it has not been possible to capture the architecture of a software project and to make it available for reuse.

Finally, to be of practical use, reuse has to be wedded to extensibility. No two projects

[1] The views expressed in this paper are those of its authors only. They do not in any way commit ESA or reflect official ESA thinking.

[2] Boehm B. (1994), *Megaprogramming*, Video tape by University Video Communications (http://www.uvc.com), Standford, California

are ever identical and hence software can never be *completely* reused: it must be modified and extended as well. This is difficult to do if the unit of reuse is the subroutine (or even an Ada package) because functionality extension implies source code changes and this, in a space application, requires the complete re-qualification of the routine/module. Moreover changes with architectural impact (eg. adding a new telecommand or a new operational mode) are difficult or even impossible to implement.

Software Frameworks

Software frameworks[3,4,5] are in the view of the authors the solution to the twin problems of reuse and extensibility outline above. They differ from other re-use technologies because they make *architectural* (as opposed to *code*) re-use possible and because they rely on object composition (and, to a lesser extent, inheritance) as functionality extension mechanisms. They are by now well-established in the area of system software (graphical user interfaces, editors, operating systems) and here it is argued that they can be equally well applied in the space field.

A framework is a collection of several single components with predefined cooperation between them. Frameworks are adapted to a particular application domain and capture an architectural design optimized for that domain. They predefine most of the overall architecture (ie the composition and interaction of its components) of a system but at the same time allow for customization by providing hooks where some of the default behaviours can be overridden. It is this possibility of flexibly reusing architecture – as opposed to simple source code fragments – while at the same time allowing behaviour tuning that makes frameworks so successful as reuse vehicles.

Figure 2 illustrates the framework concept. The unshaded area represents the architectural backbone of the application. This remains fixed in one application domain and is provided by the framework. The shaded blocks A and B are application-specific. They override or otherwise cooperate with framework objects to customize its behaviour. Thick lines represent method calls. Note the difference with applications built on module/subroutine libraries: in the latter case, the mission specific code typically *calls* the reused code, in a framework the mission specific code *is called by* the reused code.

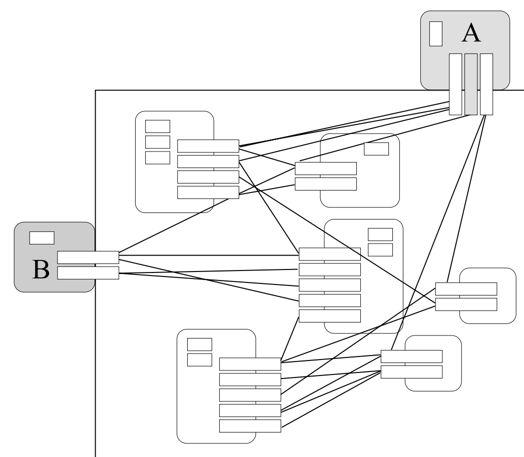

Figure 2: Framework Concept

The framework approach is well suited to situations where several functionally similar applications are developed. This is the case of the AOCS software whose overall structure changes little across projects: all AOCS implement control laws, handle telecommands, perform unit reconfiguration, etc.

OO And Real Time Systems

Frameworks are usually based on OO technology (although this need not be so: OBOSS[6] is a non-OO framework for satellite

[3] Fayad M, Johnson R, Schmidt D (1999) *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* Wiley & Sons

[4] Lewis T., Rosenstein L., Pree W., Weinand A., Gamma E., Calder P., Andert G., Vlissides J., Schmucker K. (1996) *Object-Oriented Application Frameworks*. Manning Publications/Prentice Hall

[5] Pree W (1996), *Framework Patterns*. New York City: SIGS Books

[6] fttp://ftp.estec.esa.nl/pub/ws/wsd/oboss/www/oboss.html

software) and OO technology is often regarded as unsuitable for hard real-time applications like the AOCS. The greater memory footprints and the run-time overheads of OO applications are common reasons cited to support this contention. These, however, are not insuperable hurdles[7]. Traditional AOCS systems were based on 1753 mil-std processor with a 64K address space. Future systems will probably be based on the ERC32 - a SPARC processor recently space qualified by ESA[8]. Its use dramatically increases available memory and CPU resources thus allowing to accommodate more resource-intensive technologies.

A more serious criticism of OO technology is that it is incompatible with static schedulability analysis. This type of analysis involves estimating the execution time of segments of code[9,10]. This may appear to present problems in an object-oriented system because of run-time binding of methods. Consider the following statements:

```
object = concreteObject;
. . .
object.method(arguments);
```

The estimate of the execution time of the method call depends on the (dynamic) assignment to `object` which is statically unpredictable.

Thus, static analysis is complicated by the difficulty of determining statically which concrete method is called. However, an embedded system is a closed system and the number of methods that *might* be called is finite (and, in the case of an AOCS system, likely to be quite small). It is therefore always possible to determine which is the worst-case execution time and use this estimate in the schedulability analysis. This is obviously a pessimistic estimate but pessimism is a feature of *all* methods for static schedulability.

Note moreover that run-time binding in good OO designs is often used in lieu of conditional branches and therefore the use of the worst case method will lead to the same result as would be obtained with a conventional implementation[11].

There are, however, some typical OO constructs[12,13] - commonly used in framework design - that rely on delegation of tasks along linked lists of objects with arbitrary length and these clearly defy static analysis. The difficulty they introduce is conceptually similar to that of a `for` loop with an upper bound for the loop counter that cannot be established at compile time. Such constructs can be minimized in an embedded context (the AOCS framework described below foresees their use in only two cases) and, where they are inevitable, the "closed" nature of the AOCS software always makes it possible to provide meta information to put an upper bound on the recursion level thus permitting static schedulability analysis.

Finally, it should be mentioned that the experience of the authors is that static schedulability analysis is seldom if ever used on satellite projects. Compliance with timing requirements is in practice ensured by testing and this can be done with an OO approach as well as with a conventional design.

A Software Framework for the AOCS

The concepts presented above are being tested with the development of a software framework for the AOCS. The target processor is the ERC32 processor and C++ is baselined as language. Wherever possible, use of "design

[7] D. Herity (1998), *C++ in Embedded Systems: Myth and Reality*, "Embedded Systems Programming", Feb 98
[8] http://www.estec.esa.nl/wsmwww/erc32/erc32.html
[9] L. Ko et al (1999), *Timing Constraint Specification and Analysis*, "Software - Practice and Experience", Jan. 99
[10] T. Vardanega, J. Katwijk (1999), *A Software Process for the Construction of Predictable On-Board Embedded Real-Time Systems*, "Software – Practice and Experience", Mar. 99

[11] The operational mode manager described at the end of this paper is an example of the use of run-time binding as a way of implementing a `case` switch.
[12] Examples are the "decorator" and "chain of responsibility" patterns of Gamma *et al* (see footnote 14 for full reference).
[13] W. Pree (1996), *Framework Patterns*. New York City: SIGS Books (German translation, 1997: Komponentenbasierte Softwareentwicklung mit Frameworks. Heidelberg: dpunkt)

patterns"[14] - regarded as a model of reusable design solution - is made.

In general, technological advances can be used either to expand the functionality of a piece of software or to improve its quality. Historically, the emphasis has been on the former task. This is why successive revolutions in software engineering – the transition to compiled languages, then to procedural languages, and still later to modular and OO languages – have not had a major impact on software quality (programs are as error-prone and as poorly readable/reusable today as they were in the sixties[15]!). This project concentrates on improving reuse. Hence, the AOCS framework covers only the present functionalities of AOCS systems. By constraining functionality to its present level, technological advances is leveraged to improve quality and reduce development costs.

Frameworks often suffer from being too large and monolithic. The authors advocate the *framelet approach*[16]. As the name implies, a framelet is a "small framework" (or a "large design pattern"). It formalizes a design solution to a single, domain-specific design problem. Framelets can be used in isolation or can be combined to form a framework. Because of space limitations, the remainder of this paper describes only three framelets, to handle telecommands[17], telemetry[18] and operational mode changes[19].

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

[15] F. Brooks (1995), *The Mythical Man-month - Essays in Software Engineering*, Addison Wesley Publishing Company

[16] W. Pree, K. Koskimies (1999), *Framelets – Small and Loosely Coupled Frameworks*, ACM Computing Survey Symposium on Application Frameworks, M. Fayad Publishing Company, Dec. 99

[17] W. Pree, A. Pasetti (1999) *An Active Telecommand Framelet for the AOCS Software*, Internal SWE Document Ref. SWE/99AOCS/002

[18] W. Pree, A. Pasetti (1999) *A Telemetry Framelet for the AOCS Software*, Internal SWE Document Ref. SWE/99/AOCS/003

[19] A. Pasetti, W. Pree (1999) *General Architectural Issues for an AOCS Framework*, Internal SWE Document ref. SWE/99/AOCS/004

The Telecommand Framelet

Telecommands encode actions to be performed on the AOCS software. A telecommand is a string of data bytes with the structure shown in figure 3: a header word identifies the telecommand type; the identifier is followed by one or more data words and a checksum terminates the telecommand.

Telecommands are executed on-board by a *telecommands handler* which essentially consists of a `case` statement that processes each telecommand according to its type as defined by the telecommand identifier.

This architecture has the advantages of simplicity and small telecommand size but suffers from 3 drawbacks: 1) weak expressiveness; 2) tight coupling between telecommand structure and telecommand handling software; and 3) no re-usability of software across missions.

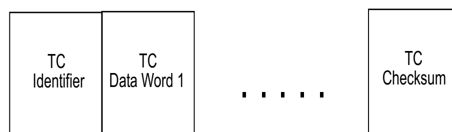

Figure 3: Current telecommand structure

The concept adopted for the AOCS framework applies the object-oriented paradigm in a manner analogous to graphical applications where graphic objects carry within themselves the code for their own representation as well as their data. A telecommand then becomes a component that exposes a method called `execute` which performs the actions associated with the telecommand itself. Using a Java-like pseudo-code notation, a telecommand is an object that implements the following interface:

```
interface Telecommand {
     void execute();
     AocsTime getTimeTag();
     int getCheckSum();
}
```

The `execute` method is called by the telecommand handler which in this way

remains completely insulated from any knowledge of what the telecommand does and how it does it. This concept presupposes an AOCS software that is built as a collection of objects that expose certain interfaces. The telecommands use these exposed interfaces to perform their assigned tasks. Methods `getTimeTag` and `getCheckSum` implement other operations common to all telecommands.

As an example, consider a telecommand that must switch on a sensor. The sensor is assumed to be implemented as a component that, among others, exposes a `switchOn` method. The telecommand's `execute` method then simply calls `switchOn` on the desired sensor component.

The chief drawback of this concept is the greater size of telecommands due to their carrying code as well as data. Greater size is offset by greater re-usability of both the individual telecommands and the telecommand handling software. Individual telecommands can be re-used because they consist of calls to public methods – whose syntax need not change from mission to mission. The telecommand handler can be re-used because it is completely de-coupled from telecommand content. In the traditional concept the handler *interprets* telecommands. In the new concept, telecommands *execute themselves* and the handler must simply trigger their execution. Telecommands and the logic required to execute them have been separated and consequently the telecommand handler has become mission-independent.

Independence of telecommand handlers from telecommand content also means that new telecommands can be added without any impact on the telecommand handling software thus improving ease of extensibility.

Telecommands are now more expressive because they can call any public method in the AOCS software whereas with the conventional concept they are limited to the commands that are understood by the handler. Their greater expressiveness can be exploited to combine into a single telecommand actions that would otherwise result in several telecommands. Consider for instance four units that have to be switched on in sequence. With the current telecommand concept, this requires sending four telecommands one after the other. With the active telecommand concept, the commands can be naturally combined into a single telecommand. This results in a reduction of overhead that somewhat compensates for the higher overhead intrinsic to the active telecommand concept.

Telecommand expressiveness finally allows telecommands to be made responsible for checking that the AOCS is in the correct state prior to their execution. A mode transition telecommand, for instance, checks that the units required in the new mode are correctly configured and can refuse to execute itself if they are not. In the passive telecommand concept this task is left to the telecommand handler that thus has to perform a large variety of telecommand-specific checks.

Finally, this telecommand concept allows to treat sequences of telecommands as *transactions*. The term "transaction" is used in the same sense in which it is used in database systems, namely it designates an atomic operation that can either succeed or fail and that, in case of failure, restores the initial state of the system. In current systems, execution failure for a telecommand is reported in telemetry and it is then left to the ground to take whatever corrective action is appropriate. A telecommand transaction is safer because, in case of execution failure, the AOCS software is left in a consistent state.

Telecommands are incorporated in transaction-like sequences by providing them with an `unExecute` operation (similar to the `undo` of most desktop application). Consider for instance an AOCS that is in mode A and must make a transition to mode B. Suppose also that mode B requires units U_1, U_2 and U_3 to be switched on. The following commands are required to perform the desired transition:

1) switch on unit U_1; 2) switch on unit U_{12}; 3) switch on unit U_{13}; 4) switch to operational mode B. If, say, the second telecommand fails, two corrective actions must be performed: 1) abort the telecommand sequence; 2) switch off unit U_1. Simply aborting the telecommand sequence would leave the AOCS in an inconsistent state where unit U_1 (which should be switched off in mode A) remains powered. When the four commands are treated as a transaction, then the failure of command 2 automatically triggers an `unExecute` for command 1 thus preserving the consistency of the AOCS state.

The Telemetry Framelet

Telemetry data are generated cyclically by the AOCS for transmission to the ground. In current implementations, the *telemetry handler* directly collects the data for telemetry and stores them in a dedicated buffer from which the data will be transferred to the central satellite computer. The telemetry handler must thus have an intimate knowledge of the type and format of the telemetry data. It is this coupling between telemetry handler and telemetry objects that makes the handler so mission-specific and hinders its re-use.

In the AOCS framework approach, the AOCS software is organized as a collection of objects and each object is potentially capable of writing itself to telemetry. In practice, this means that each object is made to implement the following interface:

```
interface Telemeterable {
      void writeToTelemetry();
      int getImageLength();
}
```

A call to `writeToTelemetry` causes the object to write its internal state to the telemetry stream. Method `getImageLength` returns the lengths in bytes of the image that is created by `writeToTelemetry`. This information is useful to the telemetry handler to check whether the object selected for inclusion in telemetry actually fits into the memory space available for telemetry.

The task of the telemetry handler becomes simply to keep track of the objects that are to be included in telemetry and to call their write-to-telemetry method. In this way, the handler does not need to know anything about the internal structure of objects and can thus be designed in a mission-independent manner.

The telemetry handler then is an instance of the following class:

```
class TelemetryHandler {
  TelemetryList tmList;
  void addToTm(Telemetarable t);
  void removeFromTm(Telemetarable t);
  void makeTmFrame();
}
```

`tmList` is a data structure that contains the sequence of objects to be included in the next telemetry frame. Methods `addToTm` and `removeFromTm` can be called to add to or remove objects from `tmList`. Typically, these methods would be called by a `Telecommand` object representing ground commands to modify the format of the next telemetry frame. Method `makeTmFrame` is called periodically to assemble a telemetry frame:

```
void makeTmFrame() {
   for (all objects t in TmList) do
      t.writeToTelemetry();
}
```

It should be clear that the telemetry handler is now completely mission-independent and can be ported from one AOCS to the next without any change at all.

Telemetry and Serialization

There is a tantalizing conceptual similarity between telemetry and serialization (in the Java sense of the word). Both processes involve the writing of objects' states to an output stream. Serialization uses standard procedures to do the writing and is thus more reusable than the concept proposed here where a dedicated method, `writeToTelemetry`, must be implemented for each AOCS class.

Serialization is at present not baselined for the AOCS framework because of the need for meta-language information[20] and because of concerns about memory overheads but, because of its elegance and generality, its use will be reassessed during the project.

The Operational Mode Framelet

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 moving towards a component-based approach, the basic design choice was between keeping a single operational mode for the whole AOCS software, or making operational mode a property of individual components.

The first approach requires a "mode manager" responsible for ensuring that each object behaves in a manner consistent with the current mode. This object acts as a centralized coordinator of object behaviour. As such, it requires a detailed understanding of the internal state of each object. This approach weakens data encapsulation and was rejected.

The selected approach makes operational mode a component-level property: components are responsible for changing their own operational mode in response to changes in their environment. Components keep track of environmental changes by monitoring properties of other objects: they explicitly register interest in external properties and ask to be notified when these properties changes in a certain manner. In fact, the property monitoring mechanism is the object of another AOCS framelet[21] loosely modelled on the JavaBeans property model[22].

[20] Meta-language information is provided as part of the language in Java but not in C++.

[21] A. Pasetti, W. Pree (1999), *General Architectural Issues for an AOCS Framework*, Internal SWE Document Ref. SWE/99/AOCS/004.

[22] R. Englander (1997), *Developing Java Beans*, O'Reilly

Note that a monitored property could also be the operational mode of another component. Hence, the traditional architecture with an AOCS-wide operational mode could be implemented by having components implement the same set of modes and by having components change their own mode to follow changes initiated by a "mode manager" object.

Components implement algorithms. The algorithm type depends on their operational mode. The mode management implementation follows a design pattern (akin to the "strategy pattern" of Gamma *et al*[23]) that separates the mode logic from the algorithm implementation. The design pattern is illustrated with an example.

Consider a `Controller` object responsible for the implementation of the attitude control law. An outline definition of its class is as follows:

```
class Controller {
  ControllerImplementer implementer;
  ControllerModeManager modeManager;

  void computeTorque() {
    implementer.computeTorque();
  }

  void run() {
    implementer:=
       modeManager.getImplementer();
      . . .
    computeControlTorque();
      . . .
  }
}
```

The `Controller` is an active object whose `run` is called periodically by the scheduler. In this example, it is assumed that the `Controller` consists of one single method, `computeTorque`, besides method `run`. The work to be done by this method is entirely delegated to an implementation object, `implementer` which contains the mode-specific implementation of `computeTorque`. Object `implementer` is re-defined at every

[23] See footnote n. 14

activation of `Controller`. Its definition is done by object `modeManager`.

The `modeManager` maintains several versions of `implementer`, each one adapted to a particular operational mode. When its `getImplementer` method is called, it checks what the current operational mode is and returns a reference to a `ControllerImplementation` of the appropriate kind. All operations of the original `Controller` are then delegated to this mode-specific `ControllerImplementer`.

This design separates implementation of the mode algorithms (confined to the various `implementer` objects maintained by the mode manager) from the mode switching logic (confined to the mode manager). Reusability is achieved though object composition that allows the construction of a mission-specific controller by plugging together appropriate `modeManager` and `Implementer` objects *without any source code changes*.

Comparison with Autocode Tools

Autocode generators are an alternative solution to the software problem. In particular, in the AOCS field, both Xmath and MatLab have successfully been applied to develop software implementing attitude and orbit control law[24]. There are important conceptual differences between them and the framework approach advocated here.

Firstly, an autocode tool does not by itself facilitate the design of the software architecture. *Given the architecture*, the tool makes it easy to generate the software but the design of the architecture still has to be done manually by the user on an *ad hoc* basis. By contrast, frameworks relieve developers of the need to define the architecture because the architecture is built into the framework (indeed, the architecture *is* the framework).

A second difference stems from the greater generality of the autocode tools that are targeted at a very wide range of users (eg. Xmath is targeted at dynamic system modelling, not just at AOCS modelling). An AOCS framework is instead specifically targeted at AOCS applications and is therefore endowed with abstractions tailored to this application domain. For instance, a generic autocode tool will not have an abstraction for 'reaction wheel' and this object must therefore be built from simpler entities for each new AOCS application. An AOCS framework would instead be likely to include a 'reaction wheel' abstraction that can be directly manipulated by users.

Thirdly, the generality of autocode tools implies that they seldom can generate more than a fraction of the total code required for the AOCS. Xmath, for instance, is good at generating code implementing a spacecraft's control laws but poor at generating code to perform functions like: telecommand handling, telemetry generation, failure detection, reconfiguration management, etc. It is these functions that normally make up the bulk of the AOCS software[25]. An AOCS-specific framework instead in principle covers all the AOCS code.

Fourthly, autocode tools are usually built on top of environments that were intended as *simulation* and *algorithm* design environments, not as design environments for the architecture of complex systems. The facilities they provide are correspondingly optimized for the former tasks making them awkward as architectural design tools. This may make model reuse very problematic: understanding a complex Xmath model can be as difficult as understanding a complex piece of code. An AOCS framework, on the other hand, is specifically designed to be portable across projects and to have a structure that facilitates maintainability, re-use and understandability.

[24] W. Dellinger, M. Salada, H. Shapiro (1999), *Application of Matlab/Simulink to Guidance and Control Flight Code Design*, 22nd AAS FNC Conference, Breckenridge, Colorado, Feb. 99

[25] A distributed architecture (common on European projects) is assumed with a dedicated processor for the AOCS subsystem.

On the other hand, the origin of autocode environments as simulation tools is also their greatest strength. An autocode system provides a seamless integration of validation and development facilities. Although a framework that is based on components can also be oriented towards integrating development and testing environments, at present a framework solution is in this respect definitely inferior to an autocode solution.

The autocode and framework approaches are perhaps best seen as complementary. Autocode tools are especially good at generating code that implements specific algorithms (ie they are good at generating individual subroutines, rather than entire systems) and frameworks typically have 'holes' for the subroutines implementing the application-specific algorithms. An AOCS framework, for instance, will have handles where users can hook the subroutines implementing, say, the controller algorithms and the state estimators. These algorithms would typically be developed and tested in an environment like Xmath. This environment could be used to generate the code implementing them and this code could then be inserted in the appropriate 'holes' in the framework. This solution optimally combine the framework and autocode approaches leveraging the specific advantages of each.

Empowering Domain Specialists

The normal procedure on satellite projects is for AOCS engineers to write user requirements for their systems but to delegate software development to a separate team of software engineers *who are usually not AOCS specialists*. This results in a situation where the heart of the AOCS system – its software – is designed, developed and tested by persons who are not domain specialists. This approach is flawed because the interface between the software engineers and their AOCS colleagues lengthens development times and introduces a potential for errors.

Autocode tools empower AOCS engineers with the skills to directly generate their software. Frameworks, too, are often supplemented with automatic code generators. Combining them with the autocode generators of dynamical simulation tools will increase the share of AOCS software that is generated without manual intervention and that can be developed by AOCS engineers without recourse to software engineers.

Conclusions

This paper has argued that OO technology can and should be applied to onboard software development and that frameworks provide the solution to the reuse problem by making *architecture* as well as *code* reusable across projects. These contentions are being tested with the development of a framework using C++ on an ERC32 target. In order to fully leverage the potential of this technology to improve software quality (as opposed to expanding its functionalities) the requirements of existing AOCS are being assumed.

The framework is being developed as a collection of framelets. This paper described the framelets for telecommand and telemetry handling and for operational mode management. These examples clearly show the re-use potential of the proposed technology. In all three cases, an architecture was designed that is both reusable and extensible to fit the specific requirements of each mission.

The promise of reusable components has been made several times in the short history of software engineering. Too often, it was not kept. Framework and component technology, however, have in commercial applications gone beyond promises making reuse a fact. There is no reason why their benefits should not be available to the space community.