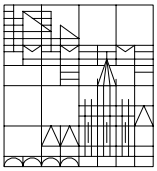


University of Constance
Dept. of Computer Science

AOCS FRAMEWORK - IMPLEMENTATION NOTES

Abstract

This document was written as part of the study "Design and Prototyping of a Software Framework for the AOCS" done under contract Estec/13776/99/NL/MV for ESA-Estec. The purpose of the study is the development of a software framework for the Attitude and Orbit Control Subsystem (AOCS) of a satellite. The framework was developed in full at the architectural design level but only a representative subset of it was implemented at the prototype level. Although there is no detailed design documentation for the AOCS framework, this document contains enough information about its implementation to allow an informed reader to understand – and where necessary modify or extend – its code.

Written By:	A. Pasetti	(University of Constance / SWE)
Date:	30 April 2002	
Issue:	2.3	
Reference:	SWE / 99 / AOCS / 020	

TABLE OF CONTENTS

1 REFERENCES

RD1 E. Gamma *et al.* (1995), *Design Patterns – Elements of Reusable Object Oriented Software*, Reading, Massachusetts: Addison-Wesley

RD2 A. Pasetti (2000), *AOCS Framework – Concept Level Description*, AOCS Framework Document ref. SWE/99/AOCS/004

RD3 Deleted

RD4 A. Pasetti (2000), *Methodological Issues*, AOCS Framework Document ref. SWE/99/AOCS/018

RD5 A. Pasetti (2000), *Inter-Component Communication Framelet*, AOCS Framework Document ref. SWE/99/AOCS/005

RD6 A. Pasetti (2000), *Object Monitoring Framelet*, AOCS Framework Document ref. SWE/99/AOCS/008

RD7 A. Pasetti (2000), *Data Processing Framelet*, AOCS Framework Document ref. SWE/99/AOCS/006

RD8 A. Pasetti (2000), *AOCS Unit Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/017

RD9 A. Pasetti (2000), *Reconfiguration Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/015

RD10 A. Pasetti (2000), *Operational Mode Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/009

RD11 T. Brown, A. Pasetti (2000), *Manoeuvre Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/012

RD12 A. Pasetti (2000), *Failure Detection Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/010

RD13 A. Pasetti (2000), *System Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/021

RD14 A. Pasetti (2000), *Failure Recovery Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/011

RD15 A. Pasetti (2000), *Telemetry Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/003

RD16 A. Pasetti (2000), *Telecommand Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/014

RD17 A. Pasetti, T. Brown (2000), *Controller Management Framelet*, AOCS Framework Document ref. SWE/99/AOCS/016

RD18 A. Pasetti (2000), *AOCS Framework – Prototype Definition*, AOCS Framework Document ref. SWE/99/AOCS/019

2 ACRONYMS

AAD	Attitude Anomaly Detection
AOCS	Attitude and Orbit Control Subsystem
AST	Autonomous Star Tracker
CSS	Coarse Sun Sensor
ES	Earth Sensor
FDIR	Failure Detection, Isolation and Recovery
FPM	Fine Pointing Mode
FSS	Fine Sun Sensor
GYR	Gyroscope
KF	Kalman Filter
IAM	Initial Acquisition Mode
MIMO	Multi-Input-Multi-Output
NM	Normal Mode
NTT	Non-Time-Tagged
OBDH	On-Board Data Handling system (aka as OBDS)
OCM	Orbit Control Mode
OO	Object-Oriented
PD	Proportional-Derivative controller
PI	Proportional-Integral controller
PID	Proportional-Integral-Derivative controller
RRM	Rate Reaction Mode
RTOS	Real-Time Operating System
RW	Reaction Wheel
SAS	Sun Attitude Sensor
SBM	Stand-By Mode
SISO	Single-Input-Sinle-Output
SPS	Sun Presence Sensor
STR	Star Tracker
SLM	Slewing Mode
SM	Safe Mode
TC	Telecommand
THU	Thruster
TM	Telemetry
TT	Time-Tagged

VS	Visual Studio

3 INTRODUCTION

This document contains some notes relative to the implementation of the *prototype AOCS framework*. The prototype AOCS framework is a partial implementation of the AOCS framework. This document is intended to serve as a guide to the implementation of the prototype in lieu of standard detailed design documentation which – owing to budget and schedule constraints – could not be generated within the AOCS framework project. The objective of this document is to give enough information to allow an informed reader to understand – and where desired to modify and extend – the prototype framework.

Where appropriate, this document also consider alternative implementations and implementation improvements.

3.1 Context

The context for the definition of the prototype framework is the architectural design of the full framework. This is presented at system concept definition level in RD2 and at framelet concept and at framelet architectural definition level in RD5 to RD17.

The prototype framework is described in RD18.

4 SOFTWARE ORGANIZATION

This section describes how the prototype framework software is organized in its full form. Please note that the standard delivery package does not include the test software (see section 7).

The software delivered with the prototype framework can be divided into four parts:

- The *Framework Software*: all the components and abstract interfaces making up the AOCS framework
- The *Framework Testing Software*: the software modules that were used to test at unit and class level the framework software
- The *Framework Instantiation Software*: the software modules and components required to instantiate the AOCS prototype
- The *AOCS Prototype Software*: the AOCS prototype application instantiated from the prototype framework

It should be noted that the framework testing software is not a formal deliverable. It is provided without documentation and its quality is inferior to that of other parts of the framework software.

The prototype framework was initially implemented in Microsoft's VisualStudio (VS) environment. Here, initial debugging and functional testing was done. In a second stage, the framework software was ported to an ERC32 simulator where performance testing was carried out. The two versions of the AOCS framework software are described in the next two subsections.

UML models were used to describe the framework architecture. The relative files are stored as described in subsection 4.3

4.1 Visual Studio Software Organization

The VS version of the framework software is organized as a single VS workspace containing 18 projects.

The directory structure of the software is illustrated in the table below where subdirectory levels correspond to indentation levels. The table only shows user-generated directories. Directories that were automatically generated by the VS environment are not shown.

Directory Name	Description
AocsFrameworkHome	AOCS Framework Home Directory
VSWorkspaces	Root directory for VS workspaces
AocsFrameworkPrototype_1	Workspace file for prototype framework
VSProjects	Root directory for VS projects
AocsData	Source files to define AOCS data concept
AocsDataTest	Project directory to test AOCS data concept
AocsEvent	Source files to define AOCS event concept
AocsEventTest	Project directory to test AOCS event concept
AocsFactories	Source files related to AOCS prototype instantiation
AocsPrototypeTest	Project directory for the AOCS prototype test
AocsUnit	Source files to define AOCS unit concept
AocsUnitTest	Project directory to test AOCS unit concept
BasicObjects	Source files to define AOCS basic classes
BasicObjectTest	Project directory to test basic classed
ControllerManagement	Source files for controller management f/l
ControllerManagementTest	Project directory to test controller management f/l
FailureDetectionManagement	Source files for failure detection f/l
FailureDetectionManagementTest	Project directory to test failure detection f/l
FailureRecoveryManagement	Source files for failure recovery f/l
FailureRecoveryManagementTest	Project directory to test failure recovery f/l
GeneralInclude	General purpose include files
GeneralUtilities	General purpose procedures
ManoeuvreManagement	Source files for manoeuvre f/l
ManoeuvreManagementTest	Project directory to test manoeuvre f/l

ModeManagement	Source files for operational mode f/l
ModeManagementTest	Project directory to test operational mode f/l
ObjectList	Source files to define object lists
ObjectListTest	Project directory to test object lists
ObjectMonitoring	Source files for object monitoring f/l
ObjectMonitoringTest	Project directory to test object monitoring f/l
OperatingSystemObjects	Source files for objects performing OS-like functions
ReconfigurationManagement	Source files for reconfiguration f/l
ReconfigurationManagementTest	Project directory to test reconfiguration f/l
RegressionTest	Project directory for the regression test
SequentialDataProcessing	Source files for data processing f/l
SequentialDataProcessingTest	Project directory to test data processing f/l
SystemManagement	Source files for system management f/l
SystemManagementTest	Project directory to test system management f/l
TelecommandManagement	Source files for telecommand f/l
TelecommandManagementTest	Project directory to test telecommand f/l
TelemetryManagement	Source files for telemetry f/l
TelemetryManagementTest	Project directory to test telemetry f/l
Xmath	Source files for interface to Xmath

Thus, the VS framework software is entirely stored under directory `VSProjects`. This software is organized in subdirectories that gather together groups of logically related files. These subdirectory are called *subsystem directories*. Typically, subsystem directories contain all the files required to define an important type in the AOCS framework or an entire framelet.

The subsystem directories only contain source files (.h and .cpp files). Executables and object files are stored in *project directories*. A project directory contains the project files for a VS project. All projects are associated to tests for some specific feature of a group of files.

The project name is always obtained as `<feature>Test` where `<feature>` is the name of the feature to be tested. Thus, for instance, the `AocsData` type is tested in a project called `AocsDataTest.`

The project directories have additional subdirectories which are created by the VS project management.

4.2 ERC32 Software Organization

The ERC32 version of the AOCS framework files are stored within the ERC32 simulator directory tree. If `/erc32` is the simulator home directory, then their base directory is: `/erc32/src/aocs-framework-1`. The directory structure below this level is the same as in the VS environment except for the `MemoryTest` subdirectory that is specific to the ERC32 environment and contains the code for performing the memory measurements.

Each subsystem directory has its own `make` file. A `make` file in `/erc32/src/aocs-framework` calls all lower level `make` files and can be used to reconstruct the entire ERC32 version of the AOCS framework software.

It must be stressed that the dependency files in the make files are not guaranteed to be complete. The only safe way to reconstruct an executable, is to delete all the object files and then call its `make` file.

Regeneration of all AOCS framework executables takes about 2-5 minutes on an ordinary SUN SPARC workstation.

The only executables that should be build in the ERC32 environment are:

- `AocsPrototypeTestErc32`
- `TimingTestErc32`
- `MemoryTestOnlyFunctMan`
- `MemoryTestFullFw`
- `EmptyMemoryTest`
- `RegressionTest`

These executables also exist in a debug version.

Make files are present to build other executables but these are not supported and are provided for information only.

4.3 UML Models Files

The UML models describing the framework architecture were generated using *Together* version 4.0. The directory structure for the *Together* files is as shown in the following table (subdirectory levels correspond to indentation levels):

Directory Name	Description
`AocsFrameworkHome`	AOCS Framework Home Directory
`Together`	Base directory for *Together* files
`Xxx`	Together project directory
`SourceFiles`	Read/Write project directory for *Together* project `Xxx`

In the AOCS framework project, *Together* was essentially used to draw class diagrams for inclusion in the framework documentation.

There are several `Together` projects (ie. several Xxx directories). Typically, to each *Together* project therefore corresponds one framelet. In general, the *Together* projects have a project directory at location `Xxx/SourceFile` where new classes are created that are needed only for drawing diagrams (ie. classes that are not part of the framework software). Additionally, the projects may use framework directories as either read-only project directories or as search directories.

5 PORTING FROM VISUAL STUDIO TO ERC32 ENVIRONMENT

As discussed in section 4, the framework software was first developed under Microsoft's Visual Studio and then ported to the ERC32 development environment.

5.1 Conditional Compilation

There are very few differences in the framework software as it runs in the VS and in the ERC32 environment. Conditional compilation was used to cover these differences. Conditional compilation is determined by the value of the `#define` symbol MS_HOST. This symbol is in turn defined in the `CompilerSwitches.h` include file. The symbol should be defined for tests under Visual Studio and should be undefined for tests in the ERC32 environment.

5.2 Compilation Inconsistencies

In general, the VS and ERC32 C++ compilers were found to be largely compatible. The few cases were inconsistencies were found are:

- In the Visual Studio environment, the following is legal:

```
void someFunction(AocsObjct* arg){
     . . .     // function definition
}
. . .
someFunction(NULL);     // function call
```

  In the ERC32 development the function call is rejected by the compiler and the constant NULL had to be explicitly cast to (`AocsObject*`).

- The Visual Studio environment has <Math.h> as a valid name for the math library. The ERC32 only accepts <math.h>

- The telemetry framelet uses the `memcpy` built-in function. In the Visual Studio environment, this function is declared in both <`memory.h`> and <`string.h`>. The ERC32 environment only accepts the latter `#include` file.

- In the Visual Studio compiler, the scope of variables defined in a *for-init-statement* is the same as the scope of the module enclosing the `for` statement. By default, in the ERC32 environment, the scope of such variables is limited to the `for` loop. If the compiler

directive `–fno-for-scope` is enabled, the ERC32 compiler behaves like the Visual Studio compiler.

- Some of the load modules for the ERC32 environment are compiled with the `–rtems` switch in order to link in the RTEMS libraries and include files. One of the RTEMS files defines the symbols FALSE and TRUE which are also defined in the Xmath include file `sa_defn.h`. This double definition causes a compiler warning. This was removed by modifying the `sa_defn.h` file to define the FALSE and TRUE symbols only when compiling in the MS_HOST environment

5.3 Other Porting Problems

The only porting problem encountered in the migration from the VS to the ERC32 environment is the following. Code that must be executed before the `main` program is entered will give rise to errors in the ERC32 environment. This situation arises typically with non-primitive static or global variables whose constructor has to be executed before the main program. The prototype AOCS implementation has to be modified to avoid this situation.

5.4 Suspected Bug in Erc32 Environment

The regression test on the ERC32 platform did not work in its original form because of a fatal address error being reported in the `TestCase` constructor on the following instruction:

```
testName = new char[nameLength];
```

The error occurred when the constructor was called by the following instruction in the `RegressionTest` main program:

```
testSuite.loadTestCase(new TestCaseControlChannel_3());
```

Investigations showed that:

- the error is not due to the system running out of heap memory;
- the error occurs on the third call to `testSuite.loadTestCase()` with a control channel test case as argument. It does not occur on calls with any different argument
- after `testSuite.loadTestCase()` has been called once with a control channel test case as an argument, then the error occurs on the second call of: `new char[24]` (24 is the length of the control channel test names).
- the error disappears if the length of the control channel names is changed from 24 to 20

It is believed that the error is due to a bug in the ERC32 environment. The error was removed by changing the length of the control channel names to 20 characters.

6 INCLUDE FILES POLICY

There are two files for each C++ class in the framework. The header file with extension `.h` contains the class declaration. The body file with extension `.cpp` contains the class definition. The file name is the same as the class name.

In order to facilitate the collection of `include` files, each subsystem directory contains a file called `include.h` that contains all the header files from that subsystem directory.

The policy adopted in respect of include files can be formulated as follows:

– A file – `ForwardDeclarations.h` – is provided containing the forward declaration of all framework classes

– Non-trivial header files always include the forward declaration `#include` file

– Header and body files directly include the `#include` files that define the types and classes they use

6.1 General Purpose Include Files

Directory `GeneralInclude` contains the following general purpose `include` files:

- `BasicType.h`

  This file gathers together user-defined types (`typedef` definition) and enumeration types.

  User-defined types are used to avoid dependency on hard-coded types. The most common user-defined type is `Real` that is used for all variables representing floating point quantities.

- `Constant.h`

  This file gathers together the definition of the general purpose constants used in the AOCS framework.

- `ClassId.h`

  Class identifiers for all the class in the AOCS framework.

- `ForwardDeclarations.h`

  Forward class declarations.

- `CompilerSwitches.h`

Compiler switch definitions used to differentiate the executables built for testing under the Visual Studio environment from the executables built for the ERC32

7 UNIT TESTING

For each framelet or subset of a framelet, a test program is provided that performs unit-level testing on the components exported by the framelet. In the VS environment, the test programs are the executables associated to the VS projects (see section 4.2). In the ERC32 environment, the test programs are built by the `make` files in the `XxxTest` subdirectories (see section 4.2).

The test programs are not a formal deliverable of the AOCS framework study and are not included in the standard framework delivery package. They are provided upon request for information only and without documentation. Owing to schedule and resource constraints, it was not possible to ensure that they have the same quality as the rest of the framework software.

The test programs are organized as *test case*. A test case is encapsulated in a class derived from the following base class:

```
class TestCase {

     . . .

public :

     TestCase(int testId, char* testName);

     // To be called before the test starts. Returns false if
     // initialization failed
     virtual bool setUpTestCase()=0;

     // To be called to execute the test
     virtual void runTestCase()=0;

     // To be called after test execution. Returns false if
     // shut down failed
     virtual bool shutDownTestCase()=0;

     // Set the test result -- can only be called once!
     void setTestResult(bool outcome, char* failureMessage);
     . . .
};
```

Test cases are completely self-contained.

Sequences of test cases can be loaded into a *test suite.* Test suites are encapsulated in instances of the following class:

```
class TestSuite {
     . . .
public :

     TestSuite(FILE* out);

     // Execute all test cases in the test suite
     void runTestSuite();

     // Load a test case in the test suite
     void loadTestCase(TestCase* newTestCase);
};
```

In the VS environment, the test routines send their output to a test file. The typical output of a successful test looks like this:

```
     Running test FailureDetectionManagementTest ...
     Test run performed on: Wed Nov 15 17:48:51 2000

     Test failureDetectionManagementTest_1 executed successfully ...
     Test failureDetectionManagementTest_2 executed successfully ...
     Test failureDetectionManagementTest_3 executed successfully ...
```

In the ERC32 environment the test output is sent to the console.

In case of failure, analysis of the test program code is necessary to establish the cause of the failure.

In the VS environment, test suites are provided for each framelet. In the ERC32 environment only the regression test is provided.

7.1 Regression Test

A regression test is provided that executes all the test cases for all the framelets. Its nominal output in the VS environment is:

```
Running test RegressionTest ...
Test run performed on: Wed Feb 07 13:52:54 2001

Test TestCaseAocsData_1 executed successfully ...
Test TestCaseAocsData_2 executed successfully ...
```

```
Test TestCaseAocsData_3 executed successfully ...
Test TestCaseAocsData_4 executed successfully ...
Test TestCaseAocsData_5 executed successfully ...
Test TestCaseAocsData_6 executed successfully ...
Test TestCaseAocsData_7 executed successfully ...
Test TestCaseAocsData_8 executed successfully ...
Test TestCaseAocsEvent_1 executed successfully ...
Test TestCaseAocsEvent_2 executed successfully ...
Test TestCaseAocsEvent_3 executed successfully ...
Test TestCaseAocsEvent_4 executed successfully ...
Test TestCaseAocsEvent_5 executed successfully ...
Test TestCaseAocsUnit_1 executed successfully ...
Test TestCaseAocsUnit_2 executed successfully ...
Test TestCaseAocsUnit_3 executed successfully ...
Test TestCaseAocsUnit_4 executed successfully ...
Test TestCaseAocsUnit_5 executed successfully ...
Test TestCaseAocsUnit_6 executed successfully ...
Test TestCaseAocsUnit_7 executed successfully ...
Test TestCaseAocsUnit_8 executed successfully ...
Test TestCaseAocsObject_1 executed successfully ...
Test TestCaseAocsObject_2 executed successfully ...
Test TestCaseAocsObject_3 executed successfully ...
Test TestCaseRootObject_1 executed successfully ...
Test TestCaseRootObject_2 executed successfully ...
Test TestCaseController_0 executed successfully ...
Test TestCaseController_1 executed successfully ...
Test TestCaseController_2 executed successfully ...
Test TestCaseController_3 executed successfully ...
Test TestCaseController_4 executed successfully ...
Test TestCaseController_5 executed successfully ...
Test TestCaseFailureDetection_1 executed successfully ...
Test TestCaseFailureDetection_2 executed successfully ...
Test TestCaseFailureDetection_3 executed successfully ...
Test TestCaseFailureRecovery_1 executed successfully ...
Test TestCaseFailureRecovery_2 executed successfully ...
Test TestCaseFailureRecovery_3 executed successfully ...
Test TestCaseManoeuvre_1 executed successfully ...
Test TestCaseManoeuvre_2 executed successfully ...
Test TestCaseManoeuvre_3 executed successfully ...
Test TestCaseManoeuvre_4 executed successfully ...
Test TestCaseObjectList_1 executed successfully ...
Test TestCaseObjectList_2 executed successfully ...
Test TestCaseObjectList_3 executed successfully ...
Test TestCaseObjectMonitoring_1 executed successfully ...
Test TestCaseObjectMonitoring_2 executed successfully ...
Test TestCaseReconfiguration_1 executed successfully ...
```

```
Test TestCaseReconfiguration_2 executed successfully ...
Test TestCaseReconfiguration_3 executed successfully ...
Test TestCaseReconfiguration_4 executed successfully ...
Test TestControlChannel_1 executed successfully ...
Test TestControlChannel_2 executed successfully ...
Test TestControlChannel_3 executed successfully ...
Test TestControlChannel_4 executed successfully ...
Test TestControlChannel_5 executed successfully ...
Test TestControlChannel_6 executed successfully ...
Test TestControlChannel_7 executed successfully ...
Test TestControlChannel_8 executed successfully ...
Test TestCaseXmathUcbBlock_1 executed successfully ...
Test TestCaseXmathUcbBlock_2 executed successfully ...
Test TestCaseXmathUcbBlock_3 executed successfully ...
Test TestCaseSystemManagement_1 executed successfully ...
Test TestCaseTelecommand_1 executed successfully ...
Test TestCaseTelecommand_2 executed successfully ...
Test TestCaseTelecommand_3 executed successfully ...
Test TestCaseTelecommand_4 executed successfully ...
Test TestCaseTelecommand_5 executed successfully ...
Test TestCaseTelecommand_6 executed successfully ...
Test TestCaseTelecommand_7 executed successfully ...
Test TestCaseTelemetry_1 executed successfully ...
Test TestCaseTelemetry_2 executed successfully ...
Test TestCaseTelemetry_3 executed successfully ...
```

In the ERC32 environment the regression test output looks slightly different because the output of `TestCaseTelemetry_1` which in the VS system is sent to file `TestTelemetryTest.txt` is sent to the output stream (the ERC32 has no file system).

8 OBJECT STATE

The *state* of an object is the set of values of all the attributes that define a particular instance of an object. The state of an object can be decomposed along three orthogonal directions:

- horizontally, in terms of its components
- vertically, in local and base state
- in terms of static / non-static components

These decompositions are discussed in the following subsections.

8.1 Horizontal Decomposition of Objects' State

In the AOCS framework, the state of an object is treated as made up of three distinct components:

- *The Resettable State*

  This component of an object's state covers those of its attributes that are regularly updated during the object's lifetime as a result of the object performing its allotted task.

  As an example, consider an object implementing a digital integrator. The resettable state of this object consists of the variables that are updated every time the integrator is triggered.

  The resettable state is often called just "state" where the context makes the usage unambiguous.

- *The Configuration State*

  This component of an object's state covers those of its attributes that are normally set during the application initialization to configure it. These attributes are used to tune the object's behaviour and are modified during the object's lifetime on an occasional basis only.

  Consider again the example of the integrator. Its configurable state would include the integration gain and any other parameters that are used by the integration algorithm.

  The configuration state is often called simply "configuration".

- *The Fixed State*

  This component of an object's state covers those of its attributes that must be defined once when the object is created and that cannot be modified during the object's life.

Typically, the fixed state includes data relative to the size of the internal buffers used by the object. Such data are used to allocate the memory required by the object but, since there is no dynamical memory allocation in the AOCS framework, their values are set once when the object is created (sometimes by the constructor) and cannot be altered later.

Other data that are normally part of the fixed state include the class and instance identifiers and the size of the telemetry image.

8.2 Vertical Decomposition of Objects' State

An object's state can be decomposed vertically as follows:

- *The Local State*

  For an object that is an instance of class A, the local state is the set of attributes that are directly defined by class A.

- *The Base State*

  For an object that is an instance of class A, the base state is the set of attributes that are inherited from the base class(es) of A.

  Thus, for instance, if A if derived from class B, then the base state of A includes the local state of B.

8.3 Static/Non-Static Decomposition of Objects' State

An object's state can be decomposed according to the storage class of its attributes as follows:

- *The Static State*

  This component of an object's state covers those of its attributes that are marked `static` and that are shared with all instances of the same class.

  References to event repositories are an example of static state attributes.

- *The Non-Static State*

  This component of an object's state covers those of its attributes that are not marked `static`.

  Most object attributes in the AOCS framework are non-static.

8.4 Control over the Resettable State

The resettable state is updated by the object itself and should normally not be changed by the clients of the object. Where appropriate, however, setter methods are exposed by the object to allow its clients to change its resettable state. The naming convention for such setter methods is: `set<AttributeName>`. Getter methods are normally provided for each setter method.

Objects that are derived from base class `AocsObject` inherit interface `Resettable`. This interface declares method `reset` that resets the resettable state of an object. Both the local and base parts of the resettable state are reset.

For a class A that is derived from a class B, method `reset` is normally implemented as follows:

```
void A::reset() {
     B::reset();        // reset base state
     localReset();      // reset local state
}
```

Method `localReset` is a private method that resets the local part of an object's resettable state.

8.5 Control over the Configuration State

The configuration state should normally be defined as part of an object's configuration at initialization time. The configuration state may be defined either by the constructor or through setter methods. The naming convention for such setter methods is: `set<AttributeName>`.

Objects that are derived from base class `AocsObject` inherit interface `Configurable`. This interface declares method `resetConfiguration` that resets the non-static part of the configuration state of an object. Interface `Configurable` also declares method `resetStaticConfiguration` that resets the static part of the configuration state of an object..

For a class A that is derived from a class B, method `resetConfiguration` is normally implemented as follows:

```
void A::resetConfiguration() {
     B::resetConfiguration();         // reset base configuration
     localResetConfiguration ();      // reset local configuration
```

```
}
```

Method `localResetConfiguration` is a private method that resets the local part of an object's configuration state.

8.6 Control Over the Fixed State

The fixed state should ideally be set by the constructor. However, this is not always possible. Some classes require default constructors (for instance to allow arrays of that class to be defined). In other cases, objects are created by abstract factories that do not have enough information to set the entire fixed state.

In such cases, definition of the fixed state has to be deferred. Three types of *initialization methods* are provided for this purpose.

Method `initialize` initializes the entire fixed state.

Methods `init<AttributeName>` initializes one single attribute.

Method `allocateMemory` allocates memory for internal data structures (typically arrays).

The initialization methods should be called only once. For objects that are derived from `AocsObject`, attempts to call them more than once result in a configuration error being generated.

8.7 Configuration Check

If the configuration or the fixed part of the state are not defined, the object cannot perform its allotted tasks.

Objects that are derived from `AocsObject` expose method `isConfigured` declared by interface `Configurable`. This method returns `false` if the configuration or the fixed part of the state are not fully defined. Note that a return value of `true` does *not* ensure that the object is properly configured. Thus method `isConfigured` can only indicate a configuration failure, it cannot guarantee a correct configuration.

It is normally not necessary to check that configuration of objects not derived from `AocsObject`. Where this is required (eg in the case of data item objects), methods with names like `isConfigured<ClassName>` are provided.

Normally, after the application initialization has been completed, method `isConfigured` should be called on all objects derived from `AocsObject`.

In the AOCS framework, objects derived from `AocsObject` register with the system manager when they are created and the system manager offers a method `isSystemConfigured` that automatically calls `isConfigured` on all registered AOCS objects and checks that they all return `true`.

9 OBJECT CONSTRUCTION AND DESTRUCTION

The term "object construction" refers to the instantiation of an object and to the initialization of the fixed part of its internal state.

The term "object destruction" refers to de-allocation of the memory used by the object.

9.1 Object Construction

Object construction is normally handled by a C++ constructor but, as discussed in section 8.6, calls to initialization methods may be required to complete the definition of the fixed part of the object's state.

A typical constructor in the AOCS framework has the following structure:

```
A::A() {           // constructor for class A

// initialize the fixed component of the local state
. . .

// reset the resettable component if the local state
localReset();

// reset the configuration component if the local state
localResetConfiguration();

}
```

Note that in C++ when the constructor for class A is executed, the constructors for all base classes of A are also automatically executed. Hence, A's constructor should only act on the local component of the state since the base component will be handled by the base constructor. This is why it is `localReset` and `localResetConfiguration` that should be called by A's constructor rather than `reset` and `resetConfiguration`.

Obviously, classes where the local resettable or configuration states are empty have no `localReset` or `localResetConfiguration` methods.

9.2 Delegation of Object's Construction to Derived Objects

In some cases, the construction of an object can only be completed by an instance of an object from a derived class. This situation occurs for the following initialization methods:

- `allocateMemory` in class `EventRepository`

- `initialize` in class `AocsData`
- `allocateMemory` in class `ControlChannelBlock`

Delegation of an object's construction to a derived object may be necessary for the following reasons:

- The delegated operation calls pure virtual methods that are only defined by the derived class (C++ does not allow pure virtual methods to be called by constructors)
- The delegated operation needs information that can only be provided by the derived classes.

9.3 Object Destruction

No actions are associated to objects' destruction in the AOCS framework and destructors are consequently never defined (except for the `AocsObject` destructor, see below).

Objects that are derived from class `AocsObject` should never be destroyed: they are created at initialization time and they exist until the program terminates or until the AOCS software undergoes a full software reset.

In order to ensure that this rule is adhered to, the `AocsObject` destructor is defined to create a failure report (using the standard failure reporting mechanism of the AOCS framework, see RD5) if it is called.

9.4 Implementation Alternatives

Most classes have some static data (typically, the telemetry image size buffer) that at present are initialized in the constructor. This introduces a processing overhead whenever there are more than one instance of the same class. This processing overhead could be removed by having a class instance counter that is used to ensure that static data are only initialized when the first instance of a class is created. This option however would introduce a memory overhead (to store the class instance counter).

9.5 Implementation Improvements

The following improvements could be implemented in a future version of the AOCS framework:

- Initially, the policy was followed to use only parameterless constructors. This policy was abandoned in the second part of the framework design. An inconsistency has arisen. This should be removed by using constructors with parameters wherever this is appropriate

- Instances of class `AocsObjects` should never be destroyed. A static check on this condition could be implemented by declaring a private destructor for this class. This however would make compilation of some test programs (where destruction of AOCS objects is permitted) impossible.
- Many classes in the AOCS framework should be singletons (in the sense of RD1). Their constructors could be modified to ensure that attempts to create more than one instance of such classes result in configuration error events being raised.

10 ITERATORS

Many classes in the AOCS framework expose so-called *iteration methods*. These are methods that allow the class's client to go through a list of objects contained in or associated to the class. There are three iteration methods with signatures like:

```
Object* first();
Object* next();
bool isLast();
```

where `Object` is the class of the objects in the list.

A typical usage of the iterator methods is in a for loop as follows:

```
for (Object obj=first(); !isLast(); obj=next())
{ . . . }
```

In order to ensure an efficient implementation that is compliant with the constraints of the AOCS framework, the iterator methods are *not* implemented using the C++ template library.

10.1 Implementation Improvements

Method `next` should only be called if `isLast` is not true. If it is called after the end of the list has been reached, the result is unpredictable and likely to be erroneous. The iteration mechanism could be improved by introducing a check on the call of next that raises an failure event if a call is made after the end of the list has been reached.

11 HELPER OBJECTS

It may sometimes happen that some default behaviour should be associated to an abstract interface (C++ pure virtual class). If multiple inheritance of implementation were allowed, this default behaviour could be coded into the class encapsulating the interface. In the AOCS framework – where only single inheritance of implementation is permitted – this is not possible. In this case, a *helper object* is used to encapsulate the default behaviour. Objects implementing the interface can then delegate implementation of many of the interface operations to the helper object.

In the present version of the framework, this situation arises for the `Reconfigurable` abstract interface. The abstract management of reconfigurations in a group of reconfigurable objects is the same for all reconfiguration managers. This abstract management behaviour was built into class `ReconfigurerHelper` (which implements interface `Reconfigurable`). Concrete reconfiguration managers can then create an instance of `ReconfigurerHelper` and delegate to it most of the management of the reconfigurations.

11.1 Implementation Improvements

Objects subjected to monitoring notify all their registered monitors when there is a change in the monitored property. The management of the notification and the registration of monitors are standard operations that are independent of the monitored class and could therefore be delegated to a helper object.

12 BASIC OBJECTS

The basic objects for the AOCS framework are `RootObject` and `AocsObject`. Their implementation is discussed in the next two subsections.

12.1 The `RootObject` Class

The `RootObject` class is defined as follows:

```
class RootObject {
     InstanceId instanceId;
     ClassId classId;
     static int instanceCounter;
protected :
     void setClassId(ClassId id) { classId = id; }
public :
     RootObject();
     InstanceId getInstanceId()   { return instanceId; }
     ClassId getClassId()         { return classId; }
};
```

This class defines the class and instance identifier and offers methods to get their values.

The instance identifier is obtained from a counter that is incremented by one every time a new object is created. Thus, the instance identifier of the n-th object to be created in the framework is n. The instance identifier is defined by the `RootObject` constructor.

The class identifier is defined by the constructor of each class through a call to the protected method `setClassId`. The values of the class identifier is taken from the `ClassId` general `include` file.

12.2 The `AocsObject` Class

An extract from the definition of the `AocsClass` is shown below:

```
class AocsObject: public RootObject,
                        public Resettable,
                          public Configurable,
                            public Telemeterable {

     static AocsClock* aocsClock;
     static FailureEventRepository* failureEvtRep;
     static ConfigurationEventRepository* configEvtRep;
     static int tmImageLength[NTMFORMATS];
     . . .
```

```
        TelemetryFormat currentTmFormat;
    protected :
        void reportFailure(AocsObject* creator,
                           EventType failureType,
                             AocsObject* location,
                                RecoveryAction* recoveryAction);
        void reportConfigurationError(AocsObject* creator,
                                      EventType configErrType,
                                        AocsObject* location);
        AocsTime getTime();
        AocsCycle getCycle();
    public :
        . . .
    };
```

The purpose of this class is twofold:

- to gather together the `Resettable`, `Configurable` and `Telemeterable` interfaces that are to be inherited by all non-trivial objects of the AOCS framework
- to define failure and configuration event handling services, and time handling services and make them available to its derived classes.

The first purpose is covered in RD2 and in section 8.4. The second purpose is described below.

Class `AocsObject` maintains a (static) reference to the failure event repository and to the configuration event repository and uses it to implement methods `reportFailure` and `reportConfigurationError`. These methods should be used by derived objects to report, respectively, failures and configuration errors. They allow uniform treatment of all such events throughout the AOCS framework.

Class `AocsObject` maintains a (static) reference to the AOCS clock and uses it to implement methods `getTime` and `getCycle` whereby derived objects can have a simple interface to the timing services of the AOCS framework.

Objects of type `AocsObjects` are intended to be never destroyed. They are created statically and should continue existing until the AOCS software is rebooted. In order to ensure that this condition is not violated, they are provided with a destructor that generates a failure event if it is ever executed.

12.3 Implementation Alternatives

The class identifier is now a variable in class `RootObject`. This means that every object carries a copy of its own class identifier. This has the advantage of confining the management of the class identifier to the `RootObject` class but it introduces a small memory overhead. A more efficient option would be to have the class identifier defined as a static member of each framework class. This however would require making method `getClassId` in `RootObject` virtual which would cause all objects derived from it to carry a virtual table pointer. This seemed undesirable and therefore `classId` was left as a private member of `RootObject`.

12.4 Implementation Improvements

The following implementation improvements can be envisaged:

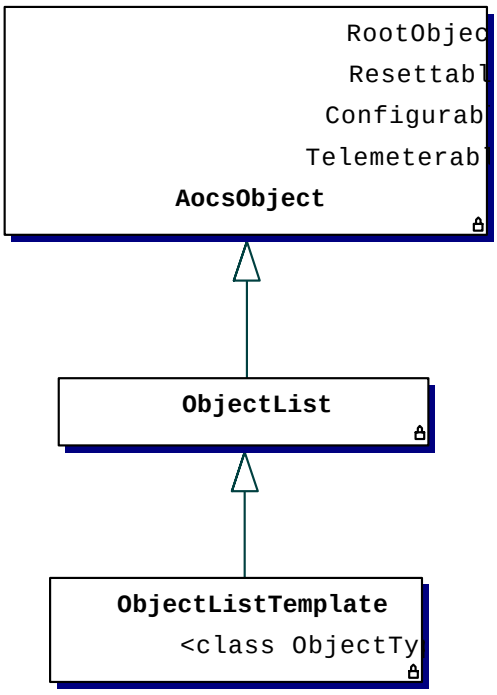
- The failure and configuration error reporting mechanisms (methods `reportFailure` and `reportConfigurationError` in class `AocsObject`) require as arguments both the originator of the report and the reference to the object where the failure or configuration error was found. In practice, it was found that the originator of the report always coincides with the location of the failure or configuration error (ie. objects find errors or failures in their own operations, not in those of other objects). Hence, one of these two arguments could be removed.
- At present, each object instance is identified by its object identifier. During testing it was found that it would be useful to be able to identify an object by a string containing a short description of the object itself. This would be useful because objects are often handled through pointers whose type, because of extensive use of dynamic typing, does not necessarily identify the object in question. Thus, an object identifier string could be associated to the instance identifier. Its associated memory overhead could be avoided by enclosing its declaration in a conditional compilation block to be disabled after testing has been completed.

13 OBJECT LISTS

Framework components often need to maintain lists of objects. For this purpose, the framework provides a generic object list of type `ObjectList`. This class manages a list of references to objects of generic type `void*`. Basically, it maintains a list of references and offers methods to add and remove references and to iterate through the references in the list.

AOCS framework components that use object lists normally need to see the list as a container for objects (or references to objects) of a specific type. If they used directly `ObjectList` they would therefore have to perform potentially dangerous cast operations.

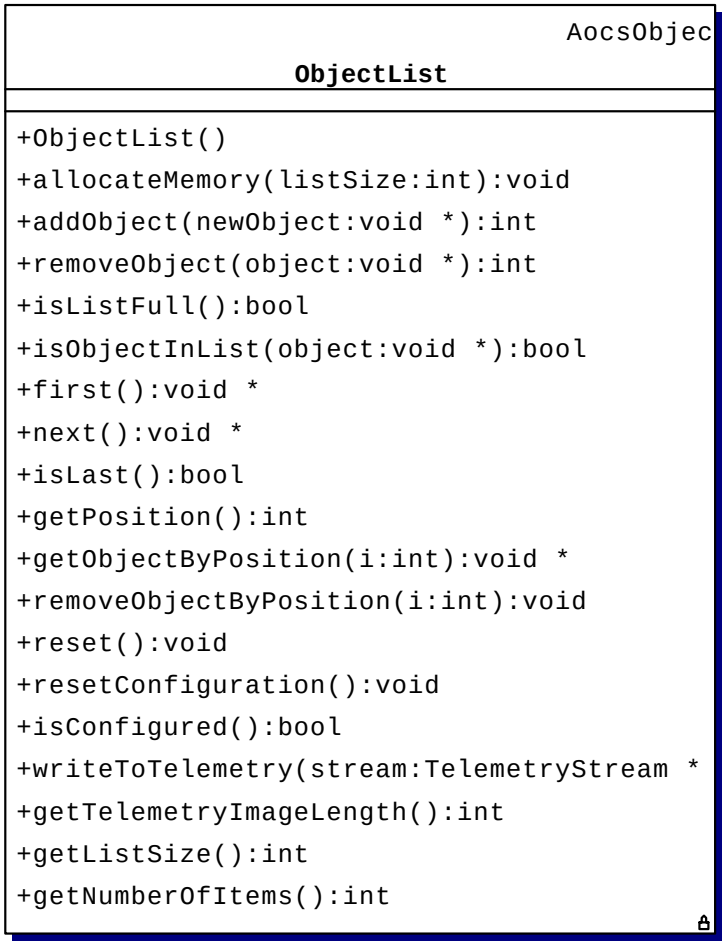
In order to avoid this danger, a template class `ObjectListTemplate` is offered that acts as a wrapper for `ObjectList` and performs the casting from `void*` to the specific type required by the client application. The resulting class structure is:



The operations in the template class are essentially implemented by delegation to the corresponding operations in the base class `ObjectList`. The only change is the introduction of casting operations to move between the `void*` type and the template type.

13.1 Object List Operations

The operations offered by class `ObjectList` are:



The public methods specific to the `ObjectList` class (ie not inherited from base classes) are described in the table:

`allocateMemory(n)`	
	Define the size n of the list and allocates the memory for the internal data structures holding the list references. This method can only be called once. Attempts to call it more than once result in configuration error events being raised.
`addObject(&object)`	

Loads object `object` into the list. There is no guarantee as to where in the list the object will be placed. The method returns an integer i in the range [0..listSize-1] that indicates where in the list the object was inserted. If the object cannot be inserted because the list is full, a failure event is raised and a negative number is returned.
`removeObject(&object)`
Methods to remove an object from the list. The method returns the position in the list of the object that was removed as an integer i in the range [0..listSize-1. If the object is not found in the list, a failure event is raised and a negative value is returned.
`isListFull`
Returns `true` if the list is full.
`first, next, isLast`
Iteration methods that iterate through the non-null references in the list.
`getPosition`
While an iteration is under way, the method returns the position of the last element retrieved by the iterator operations. The position is returned as an integer i in the range [0..listSize-1].
`getElementByPosition(i), removeElementByPosition(i)`
Returns or remove the i-th element in the list. The list elements go from 0 to (listSize-1).
`reset`
Resets any on-going iteration.
`resetConfiguration`
Resets the references to the recovery actions (see below) and clears all entries in the list.
`isConfigured`
Returns true if memory for the list was already allocated (ie. if method `allocateMemory` has been called).
`writeToTelemetry(TmStream), getTelemetryImageLength`
Telemetry methods (not implemented in framework prototype).
`getListSize`

	Returns the size of the list.
`getNumberOfItems`	
	Returns the number of items currently in the list.

13.2 Implementation Improvements

When a new object is added to the list, method add performs a linear search through the list to find the first empty slot where the object can be inserted. Similarly, when an object is to be removed, method remove performs a linear search through the list. Linear searches can be expensive in terms of processing time[1]. Future versions of object list could implement more sophisticated search mechanisms.

13.3 Implementation Alternative

The implementation proposed here wraps a template class around the basic `ObjectList` class. The wrap is required because `ObjectList` only deals with references to `void*` whereas users want to see references to specific types. With this approach, the framework code then contains a single copy of `ObjectList` but several copies of `ObjectListTemplate`, one for each type for which the template is instantiated. The overhead introduced by the multiple copies of `ObjectListTemplate` is minimal because this class, being merely a wrapper class, is very small.

The alternative approach was to make `ObjectList` itself a template class parameterized by the type of the objects to be stored in the list. This class however is rather heavy and therefore the overhead introduced by the multiple instantiation of the template would have been considerable.

[1] This problem should not be overstated: in AOCS systems as they are now, it is unlikely that more than 20-30 objects would ever be stored in an object list.

14 CLASS AOCSDATA

Class `AocsData` is the base class for the AOCS data types in the AOCS framework. It is one of the main framework hot-spots and application developers are expected to construct new classes derived from it encapsulating application-specific data types.

One of the chief concerns driving the implementation choices for class `AocsData` was the need to make derivation of these application-specific AOCS data classes as easy as possible. This meant that an attempt was made to concentrate as far as possible implementation complexity in the base class `AocsData`.

The main consequence of this decision is that many `AocsData` operations are parameterized by the number of data items in the AOCS data structure. This number is obviously undefined at `AocsData` level and varies from one application type to another (it is for instance 4 for quaternions and three for 3-dimensional vectors). Hence, `AocsData` defines a pure virtual method `getNumberOfItems` that must be defined by every subtype to return the number of data items in the AOCS data structure.

Unfortunately, since `getNumberOfItems` is a pure virtual method, it cannot be used in the constructor. Hence, some operations that logically belong to the constructor of `AocsData` must be delegated to its derived classes. The implication is the `AocsData` constructor does not follow the pattern outlined in section 8.4: its `localReset` and `localResetConfiguration` methods must be called by the constructors of its derived classes.

Class `AocsData` should also allocate memory for its internal data structures. However, the size of these data structures depends on the specific type of application data and hence the actual memory allocation must again be deferred to the constructor of the derived type. Class `AocsData` provides a method – `initialize` – for this purpose.

Class `AocsData` exposes metrics operations. The default implementation provided by this class assumes a Euclidean distance. Derived classes that need to use a different type of distance should override methods `distance`. All the other metrics methods however can remain unchanged since they are all built upon `distance`.

The AocsData implementation makes derivation of new data types very easy. At its simplest, the only piece of extra implementation to be provided by a derived class is a constructor and a (trivial) implementation for `getNumberOfItems.`

15 THE SYSTEM MANAGER

The system manager maintains a list of references to objects of type `AocsObject`. This list is implemented using the object list mechanism of section 13. Normally, it is intended to include most of the objects of type `AocsObject` instantiated by the AOCS software.

A default mechanism is provided to load the list maintained by the system manager as described in RD13: class `AocsObject` holds a reference to the system manager in variable `systemManager`. Its constructor contains the following code:

```
AocsObject::AocsObject()
{
     . . .
     if (systemManager!=NULL)
          systemManager->add(this);    // load this object in the
                                       // system manager list

     . . .
}
```

This ensures that all objects of type `AocsObjects`, created after the system manager was loaded into `AocsObject`, are automatically loaded into the system manager.

Thus, a sample initialization sequence for the AOCS software could be as follows:

```
1    // Create and configure the syst. manager and the syst. evt repository
     SystemEventRepository systemEvtRep;
     systemEvtRep.setRepositorySize(4);
     SystemManager systemManager(100, &systemEvtRep);
5
     // Create and configure the failure and config. event repositories
     FailureEventRepository failureEvtRep;
     failureEvtRep.setRepositorySize(4);
     ConfigurationEventRepository configEvtRep;
10   configEvtRep.setRepositorySize(4);

     // Create the AOCS clock
     ErcAocsClock aocsClock;

15   // Configure the static part of AocsObject
     AocsObject::setAocsClock(&aocsClock);
     AocsObject::setFailureEventRepository(&failureEvtRep);
     AocsObject::setConfigurationEventRepository(&configEvtRep);
     AocsObject::setSystemManager(&systemManager);
```

```
20   // Create other objects
     . . .
```

The system manager is loaded into the `AocsObject` class only at line 19. This means that all the objects that were created before line 19 were *not* loaded into the system manager. Objects created afterwards, on the other hand, are automatically loaded into the system manager.

It normally makes sense *not* to load the main event repositories in the system manager because this ensures that they will survive a system reset thus preserving crucial event information.

The mechanism outlined above is very convenient as it relieves the programmer of the burden of manually loading newly created objects into the system manager but it can lead to a slightly inefficient execution of a software reset. This is because calls to method resets are propagated by an object to the objects with which it has a relationship of strong association. Hence, in the event of a system reset, the latter objects will have their reset method called twice: once directly by the system manager and once indirectly by the objects that (logically) own or enclose them.

This overhead can be avoided by manually loading objects into the system manager: the programmer can then take care not to load objects that are logically contained or owned by other objects.

16 CONTROL CHANNELS

The main issues required for an understanding of how control channels are implemented are: the linking mechanisms for control channels, the internal data buffers in control blocks, the propagation of operations within super blocks, and the Xmath interface.

16.1 Control Channel Linking

Control channels are always linked at the level of the `AbstractControlChannel` interface, ie. they see other abstract control channels and do not know whether they are dealing with control blocks or control super blocks.

16.1.1 The `ccLinks` array

All control channels need to keep track of the control channels that may be linked to their inputs. This information is maintained in array `ccLinks`: `ccLInks[i]` contains a pointer to the control channel linked to the i-th input. This information is used to propagate methods calls from one control channel to the control channels that are linked to its inputs.

Note that `ccLinks` is not sufficient to fully define the input link: `a.`ccLinks[i] is a control channel which has one of its outputs linked to the i-th input of control channel `a` but the array does specify *which* input is linked to the i-th input of `a`.

Note also that some or all of the elements of `ccLinks` may be null since not all inputs of a control channel need be connected to the outputs of other control channels.

16.1.2 Control Blocks Input Links

The input links of a control block are stored in array `inputLink`. If `cc` is a control block, then `cc.inputLink[i]` is a `DataItemRead` variable that encapsulates the reference to the i-th input of cc.

16.1.3 The `IoLink` Structure

Super blocks use the `IoLink` structure to store an internal link. An internal link is a link from a super block to a control channel that is embedded in the superblock and that is linked to an input or an output of the superblock.

Note that superblocks only have direct links to those of their embedded control channels which are linked to their inputs or to their outputs.

The `IoLink` structure is defined as follows:

```
struct IoLink
```

```
{    AbstractControlChannel* cc;
     int n;
};
```

Each super blocks maintains two arrays of type `IoLink`: `inputLink` and `outputLink`. If `sp` is a superblock, then `sp.inputLink[i].cc` is a pointer to a control channel whose input number `sp.inputLink[i].n` is internally connected to the i-th input of `sp`.

Similarly, `sp.outputLink[i].cc` is a pointer to a control channel whose output number `sp.outputLink[i].n` is internally connected to the i-th output of `sp`.

16.2 Internal Control Blocks Data Buffers

Control blocks maintain four internal data buffers:

- `input, output` and `state`

  These are scratch arrays where the input, output and state values are stored during state and output propagation. These are the arrays upon which routine `propagationAlgorithm` operates. values *during* the propagation computation may be inconsistent.

- `outputBuffer`

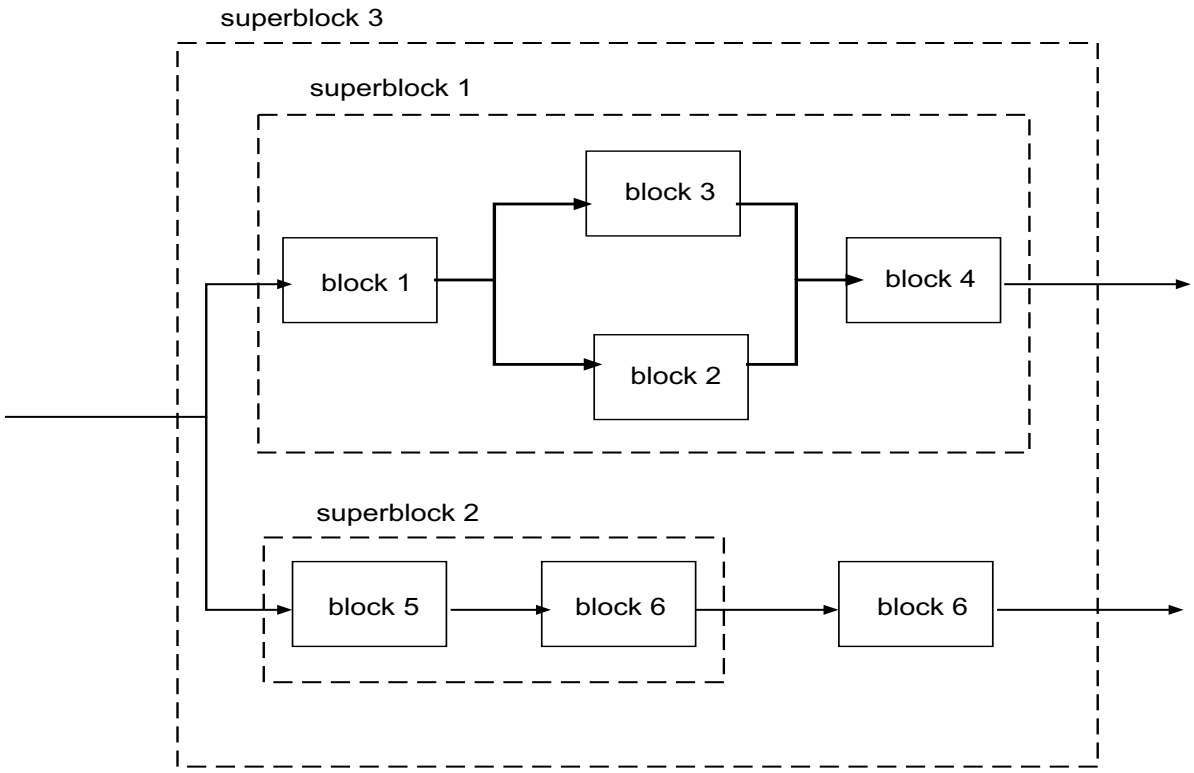
  This is an array where the last set of computed outputs is copied after `propagationAlgorithm` has terminated execution.

  This array is only useful in case preemptive scheduling is used as it allows a consistent set of outputs to be accessed even while state and output propagation is under way.

  Variable `validityTime` stores the validity time for the values stored in `outputBuffer`.

16.3 Propagation of Operations within Control Channels

Control channels can be embedded within each other. Some operations when performed on an enclosing control channel must be propagated to all control channels embedded within it. Consider for instance the following figure:



When a call to method `reset` is performed on super block 1, all the control channels within super block 1 should be reset. Note that some of the these enclosed control channels could themselves be super blocks enclosing other lower-level control channels.

The operations that are propagated from an outer control channel to the enclosed control channels are:

- `reset` : when a super block is reset, all the control channels it contains should also be reset.
- `resetConfiguration` : when the configuration of a super block is reset, the configuration of all the control channels it contains should also be reset.
- `hold/release` : when a super block is put in the hold state, all the control channels it contains should also be put in the same state. The same should happen when the super block is released.

- `writeToTelemetry` : the telemetry data of a super block are the telemetry data of the control channels it contains. Hence, calls to `writeToTelemetry` on a super block must trigger calls to the same method on all lower-level control channels.
- `getTelemetryImageLength` : because of the previous bullet, the size of the telemetry image of a super block is computed as the sum of the sizes of the telemetry images of all lower level control channels. Hence, calls to `getTelemetryImageLength` must be propagated to all lower level blocks.
- `setTelemetryFormat` : a super block and the channels it contains should have the same telemetry format. Hence, a call to setTelemetryFormat on a super block should trigger calls to the same method with the same format value on all lower-level control channels.

A general mechanism as described below is used to propagate operations from one control channel to all enclosed control channels.

Each control channel holds a reference to its enclosing control channel. This reference can be retrieved through methods `getEnclosingControlChannel`. If the method returns NULL, then the control channel is not enclosed in any other control channel.

The enclosing control channel is defined, with method s`etEnclosingControlChannel`, when the control channel is configured at initialization time. It is the responsibility of the developer to ensure that it is correctly set.

Each control channel holds a list of references to all the control channels that are connected to its inputs. This list is held in array `ccLink`. The array is automatically loaded when the inputs of the control channel are linked to external control channels.

Calls to propagation methods can be propagated backward by using the information in array `ccLinks`.

Only super blocks can act as containers for other control channels. Super blocks hold references to the control channels that are internally connected to their outputs.

When a call to a propagation method is made on a super block, the super block transfers the call to the control channels connected to its outputs. These will then propagate it backward using the `ccLink` information. The propagation stops when the beginning of the super block is reached and this is checked using the `enclosingControlChannel` parameter.

This mechanism allows a method call to be propagated (recursively, if necessary) from a super block to all the control channels it contains. However, it cannot guarantee that the operation is called only once on each control channel.

Consider for instance again super block 1 in the above figure. A call to its reset method would be propagated twice to block 1, once backward from block 2 and a second time backward from block 3.

In order to ensure that operations are called only once, a token mechanism is used. To each propagated operation is associated a unique token. In practice, this could be the time when the operation is called. A control channel holds a token value in variable `propagationToken` for which getter and setter methods are provided.

When a propagated operation is performed on a control channel, the control channel receives the token associated to that operation. An operation is performed on a control channel only if the control channel has not yet received its token. This ensures that the operation will not be performed more than once.

For an example of the implementation of this mechanism, see the implementation of the `reset` method in super blocks.

16.4 The Xmath Autocode Interface

Class `XmathUcbBlock` acts as a wrapper for UCB routines generated by the Xmath autocode tool using the default template.

The link with the UCB routine is made through a function pointer to the UCB routine. The pointer is passed to class `XmathUcbBlock` as one of its constructor parameters and is stored in variable `ucbHook`.

The constructor of `XmathUcbBlock` additionally specifies the number of inputs, outputs and state variables of the procedure superblock from which the UCB routine was generated, its sampling time and the number of real and integer parameters. The latter two parameters should always be zero in the present implementation of `XmathUcbBlock.`

The management of the `iinfo` and `rinfo` control flags for the UCB routine is difficult to understand because the Xmath documentation is not very clear on this point. The policy followed in the AOCS framework is based on the following assumptions:

- flag `iinfo[0]` is used as an error flag that can be set by the UCB routine if it finds any internal errors. The framework wrapper checks this flag after every call to the UCB routine and, if it finds it set, it raises a failure event
- flag `iinfo[1]` is an initialization flag that must be used jointly with flag `iinfo[3]` according to the following convention:
    - `iinfo[0] == 1 && iinfo[3] == 1`

This combination ensures that the UCB routine will call `init_application_data` (initialize any %variables that are used in the UCB routine) and will initialize the internal state buffers.

- `iinfo[0] == 1 && iinfo[3] == 0`

  This combination ensures that the UCB routine will call `init_application_data` (initialize any %variables that are used in the UCB routine) and then exit.

- `iinfo[0] == 0`

  This combination ensures that the UCB routine performs both state and output propagation without calling `init_application_data`.

- flag `iinfo[2]` is not used by the UCB routines.
- flag `rinfo[0]` is the time to which the outputs must be propagated in the current call of the UCB routine (ie. this is the same as the argument with which method `propagate` is called).
- flag `rinfo[1]` is the sampling time.

There are three problems with the present implementation of XmathUcbBlock which arise when the Xmath model contains so-called "percent variables" (ie. parameters that can be set dynamically to tune the behaviour of the Xmath model). The framework UCB wrapper can offer handles to allow these parameters to be set dynamically. However, their initial value is hardcoded in the autocode software (in routine `Init_Application_Data`) and when the UCB wrapper is reset, then the block parameters are reassigned their initial default values. These values can of course be changed again but only after the UCB model has gone through one iteration.

Additionally, the "percent parameters" are only accessible through the name they were given in the Xmath autocode. A better solution would be to have access to them through a generic array of parameters.

Finally, the “percent parameters” are declared as static variables in the Xmath-generated C module. This modules contains both the variable declaration and the routine implementing the Xmath model transfer function. This effectively means that the Xmath code is non-reentrant because different instances of `XmathUcbBlock` necessarily share the same set of “percent parameters”.

It is unclear whether modification of the autocode template would allow these drawbacks to be remedied.

16.5 Implementation Improvements

The following improvements could be implemented in a future version of the AOCS framework:

- Remove the fourth and firth parameters (number of integer and real parameters) of the constructor of class `XmathUcbBlock` (these parameters must always be zero).
- The propagate method of control channels instantiated from class `XmathUcbBlock` should be called at constant interval since they represent discrete procedure super blocks from the Xmath environment that are intended to be triggered at a fixed sampling rate. A check could be introduced in the propagate method of this class that verifies whether this constraint is verified and, if it is not, it raises a failure event.

17 EVENT REPOSITORIES

Concrete event repositories are derived from class `EventRepository`. This class implements all the functionalities required by event repositories except for the allocation of the memory to hold the events in the repository.

Class `EventRepository` sees the events in the repository as references to the generic class `AocsEvent`. In particular, it holds an array list declared as follows:

```
AocsEvent** list;
```

`list` thus is an array to references to `AocsEvent`. This array holds the references to the events in the repository. All repository operations are implemented using the elements in this array.

The space to hold the events themselves (as opposed to their references) is allocated by the concrete event repository classes. Consider for instance the failure event repository. Its corresponding class `FailureEventRepository` adds to its base class `EventRepository` the declaration of the following array:

```
FailureEvent* buffer;
```

`buffer` is the array that holds the events in the repository. The operations defined by class `FailureEventRepository` ensure that the two arrays – buffer in class `FailureEventRepository` and list in class `EventRepository` – remain consistent with `list[i]` holding a pointer to `buffer[i]`.

17.1 Iteration through Event Repositories

Operations `latestXxxEvent`, `previousXxxEvent` and `isLast` iterate through the events in the event repository for events of type `Xxx`. The iterators return references to the events in the repository starting from the most recent one.

The iterator operations in the concrete event repository classes are built on top of the similarly-named iterators offered by the base class `EventRepository`.

The iterators are implemented in such a manner that creation of events while an iteration is under way does not disrupt the iteration itself. It should however be noted that events created after an iteration has begun will be returned at the end of the iteration.

17.2 Event Types

The event types are stored as `#define` variables in the file `EventTypeId.h` in the subsystem directory `AocsEvent`.

18 PROPERTY IDENTIFIERS

Suppose that an object exposes n properties. The property identifier used to identify the i-th property object is derived according to the following formula:

```
propertyId = i*MAX_INSTANCE_ID+instanceId
```

where i is MAX_INSTANCE_ID is an integer defining the maximum possible value for the instance identifiers and `instanceId` is the instance identifier of the object containing the property.

For greater flexibility, an `include` file (`PropertyId.h`) is also provided that can be used to define property identifiers individually.

19 THE MODEMANAGER CLASS

The following notes help understand the implementation of the ModeManager class:

- The implementers are stored as references to type RootObject in array implementerArray. the memory for the array is allocated by the constructor to match the number of strategies and modes requested for each instance of the mode manager.
- The mode managers expose their mode as a bound property object and allow property monitors to register their interest in changes in its value. The mode manager maintains the references to the registered property monitors and their associated change objects in arrays `monitorList` and `changeObjectList`. Insertion and removal into these two arrays are always done in parallel so as to ensure that the content of the two arrays remains synchronized.
- Protected method `changeMode`(i) will cause the mode to be updated from the current mode to the target mode i. This method takes care of checking whether any external monitors should be notified of the mode change. Mode changes should always be performed through calls to `changeMode`.
- Internal mode variables use the convention that the mode indicator is an integer in the range [0..N-1] where N is number of modes. Similarly, strategy indicators are integers in the range [0..S-1] where S is the number of strategies.
- Private method `isModeLegal` is provided to check that a certain integer is a legal mode indicator (ie that it lies in the range [0..N-1]). This is declared as a virtual method to allow derived classes to use different ranges for the mode indicators. There is no analogous method for the strategy indicators since it is not anticipated that strategy indicators might have a range different from the default one [0..S-1].
- The mode manager needs to interact with the mode event repository (to log mode changes) and with the change event repository (to log changes in its mode property when the latter is monitored by external monitors). For this purpose it maintains two static references to `ModeEventRepository` and `ChangeEventRepository`.

20 THE FAILURE DETECTION MANAGER

While coding the failure detection manager (class `FailureDetectionManager`), it was realized that a type cast from class `ConsistencyCheckable` to class `AocsObject` is required.

This situation arises as follows. The failure detection manager performs consistency checks on all the objects contained in list `consistencyCheckableList`. This list contains references to type `ConsistencyCheckable`. When a consistency check fails (ie, when method `doConsistencyCheck` returns `false`), the failure detection manager must create a failure event to report the failure. One of the parameters to the supplied for the creation of the event is the "location of the failure", namely the object where the failure was found. In this case, the failure location is the object on which the consistency check was performed. This object is seen by the failure detection manager as of type `ConsistencyCheckable` but the event creation operation requires a reference to an `AocsObject`. Hence the need for the type cast.

Type casts of this kind – because they cannot be guaranteed *a priori* to be safe – should be avoided in the AOCS framework. In this particular case, the problem is not serious because only non-trivial objects implement consistency checks and all such objects are derived from `AocsObject`. However, a future implementation of the framework should be designed to avoid the need for this type of cast.

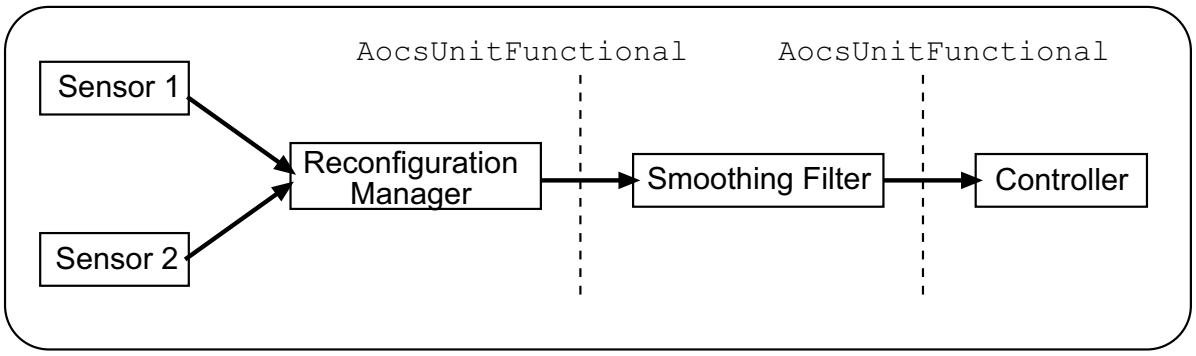
21 AOCS UNITS

This section discusses implementation issues related to the AOCS unit framelets.

21.1 Data Items and AOCS Units

AOCS units may need two types of links to data items in data pools: links to the data source buffers and links to the data destination buffers. The links are set up using methods `linkXxxOut` (link for outgoing data to source buffer) and `linkXxxIn` (link for incoming data to destination buffer) where '`Xxx`' can refer either to functional or to housekeeping data. As usual, the link is made through data item objects.

In principle, the link to source buffer is a read-only link and therefore could be done through `DataItemRead` objects. The link methods however always use `DataItemWrite` objects. This was found to be necessary when units are linked in chains of fictitious units as in the figure:



Both the reconfiguration manager and the smoothing filter components implement the `AocsUnitFunctional` interface and are therefore (fictitious) AOCS units. Clearly, the source buffers of the reconfiguration manager component must be linked to the hardware output buffers of the smoothing filter.

From the point of view of the smoothing filter, the reconfiguration buffer is a plug-in component. When it receives the reconfiguration manager, the smoothing filter must be able to access its source buffers as data item write objects because it needs to link them to its (writable) output buffers. For this reason, the reconfiguration manager must see its source buffer as `DataItemWrite` variables.

This situation is quite general. All AOCS units can potentially be combined with each other in chains of fictitious AOCS units and each unit must be able to pass its source buffer to the next

unit in the chain as a writable object. Hence it was necessary to model both the destination and source buffers in AOCS units as `DataItemWrite` objects.

For a concrete example of this situation in the prototype framework, see the `TorquingThruster` component that is combined in a chain of fictitious units with the `SapPrototype` component.

21.2 Interface to Unit Data Converters

Unit data converters are used to convert unit data from raw data level to AOCS data level. They are implemented using the control channel construct exported by the sequential data processing framelet. Control channels take as inputs data of type `Real` and produce outputs of the same type. More precisely, they take as inputs and generate as outputs `ReadDataItem` objects that encapsulate a reference to a `Real`.

In order to encapsulate unit data converters in control channels, class `AocsUnit` defines the hardware buffers as arrays of `Real`. Unit raw data, however, are often best represented as sequences of bits. The AOCS prototype for instance assumes a MACS-based interface to external units where raw unit data are most naturally represented as 16-bit unsigned short integers. In such cases, the actual unit datum is mapped to a subset of the bits making up the `Real` datum and the unit data converter will have to operate on those bits only. This obviously implies that concrete AOCS unit objects and their data converters agree on a convention to map raw data bits to a subset of the bits in the `Real` variable representing the unit hardware buffer.

In the case of the units defined for the AOCS prototype, the mapping is done as follows. Consider for instance incoming functional data. Their hardware input buffer is defined as follows:

```
Real* fcHwInBuf;                        // declaration
. . .
fcHwInBuf = new Real[nFcInpBuf]; // in the AocsUnit constructor
```

where `nFcInpBuf` is the number of functional data to be acquired from the unit. The datum actually acquired from the external unit consist of 16 bits. Such data are handled internally to the unit hardware object as variables of type `unsigned short int`. The link between the two types of variable – the `Real` and the `unsigned short integer` – is as follows:

```
unsigned short int* unitDatum;
. . .
```

```
unitDatum = (unsigned short*)&fcHwInpBuf[i];
```

The data converter that processes incoming functional data sees its input as a Real but it can simply recover the original 16-bit datum as follows:

```
Real* input;       // control channel input
unsigned short* rawDatum;
. . .
rawDatum = (unsigned short*)&input[i];
```

after the data conversion, variable `rawDatum` contains exactly the same bit pattern as variable `unitDatum` that was acquired from the MACS bus.

CONTROLLER MANAGER

The follower controller mode manager resets the controllers as they are switched in. The procedure that does this (getControllerList) assumes that all controller lists have been loaded in the same order (ie. the first controller in the list is always the Xaxis controller, the second one is always the Yaxis controller, etc).