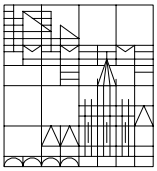


University of Constance
Dept. of Computer Science

SEQUENTIAL DATA PROCESSING FRAMELET

Concept And Architecture Description

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 will be built as a collection of framelets. This document describes the sequential data processing framelet. This framelet proposes a design pattern to handle sequential processing chains. It defines a standard interface for handling data processing chains and provides an easy way to combine data processing blocks.

Written By: A. Pasetti

Date: 30 April 2002

Issue: 2.2

Reference: SWE/99/AOCS/006

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 A. Pasetti (2000), *Inter-Component Communication Framelet – Concept and Architecture Description*, AOCS Framework Document ref. SWE/99/AOCS/005

RD4 Deleted

RD5 MatrixX 6.1.3 On Line Documentation

RD6 A. Pasetti (2001), *Software Frameworks and Embedded Control Systems*, LNCS Series, Springer-Verlag, To appear in Dec. 2001

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
OBDH	On-Board Data Handling system (aka as OBDS)
NM	Normal Mode
NTT	Non-Time-Tagged
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
SPS	Sun Presence Sensor
STR	Star Tracker
SLM	Slewing Mode
SM	Safe Mode
TC	Telecommand
THU	Thruster
TM	Telemetry
TT	Time-Tagged

3 INTRODUCTION

This document describes the sequential data processing framelet for the AOCS framework. The framelet is described at both the framelet concept level and at the framelet architectural level.

This framelet proposes an architectural solution to the problem of processing data in the data flow subsystem of the AOCS software. It defines a standard interface for processing blocks and a way to combine individual blocks into nested chains of blocks.

The provision of a standard interface for data processing blocks enhances reusability because it makes components independent of the data processing algorithms.

3.1 Context

The context for the design of the framelet is described in RD2. The present document assumes that the reader is familiar with RD1 and in particular with the overview of sequential processing chains.

RD2 identified three architectural options for sequential data processing chains:

- Data Converters
- Control Channels
- Formal Language-based Solution

The first option was abandoned because it was limited to linear processing chains with a well-defined first or last stage.

The third option is not considered here as unnecessarily complex. It may be taken up again in a later iteration of the AOCS framework design.

The second option was retained and is the one that is presented in this document.

In comparing the present document with RD2, readers should bear in mind that the class definitions presented in the latter document are not necessarily entirely consistent with the class definitions presented here. This is because the main purpose of RD2 was to introduce an architectural *concept* whereas the main purpose of the present document is to describe an architecture. The design presented here therefore should be regarded as an evolution of the design presented in RD2.

3.2 Applicability to Java Version

The AOCS Framework was first implemented in C++ and then ported to Java. This document was originally written for the C++ version and is only partially applicable to the Java version. Generally speaking, the description of the framelet at design level – in particular its design patterns – is language-independent and is equally applicable to both the C++ and Java versions whereas the architectural-level description is more tied to the C++ version. For a detailed description of the architecture of the Java framework, readers should refer to the JavaDoc documentation generated from it.

The porting of the AOCS Framework to Java was done in the "Real Time Java Project". The issues that should be borne in mind when using this document for the Java version of the AOCS framework are presented in the project web site currently located at the following address: www.aut.ee.ethz.ch/~pasetti/RealTimeJavaFramework/index.html. Some specific points to note are:

- The mechanism to link control channels to their inputs and outputs is different. It is no longer based on the data item concept that was not carried over to the Java version of the framework. It is instead based on the *data sink* and *data source* concept.
- The Java framework has an interface to Matlab autocoded routines as well as to Xmath autocoded routines.

3.3 Notation

The pseudo-code examples in this document use a C++ notation.

The class diagrams use UML notation generated with the reverse engineering tool of the *Together* tool.

4 FRAMELET CONSTRUCTS

The architectural constructs exported by this framelet are listed in the following table:

DATA PROCESSING FRAMELET
Framelet Design Patterns
Control Channel Pattern: pattern to allow uniform treatment of control channel blocks and superblocks
Framelet Interfaces and Abstract Base Classes
`AbstractControlChannel` : interface for control channels `ControlChannelBlock` : abstract class encapsulating control channel block `XmathUcbBlock` : abstract class offering an interface to Xmath autocode
Framelet Core Components
`ControlChannelSuperBlock` : container component for a control channel super-block
Framelet Default Components
`P_Block` : control block implementing a proportional transfer function `I_Block` : control block implementing an integral transfer function `D_Block` : control block implementing a derivative transfer function `AdderBlock` : control block to add two inputs `DifferenceBlock` : control block to take the difference of its two inputs `LimitBlock` : control block to saturate an input `PassThruBlock` : control block with unitary transfer function `SplitterBlock` : control block to split a single input into several identical outputs `TwoByTwoMatrixBlock` : control block implementing a 2x2 matrix multiplication transfer function `XmathUcbBlock` : control block embedding a generic UCB routine from the Xmath autocode `XmathUcbPidBlock` : control block implementing a PID controller (from Xmath autocode)

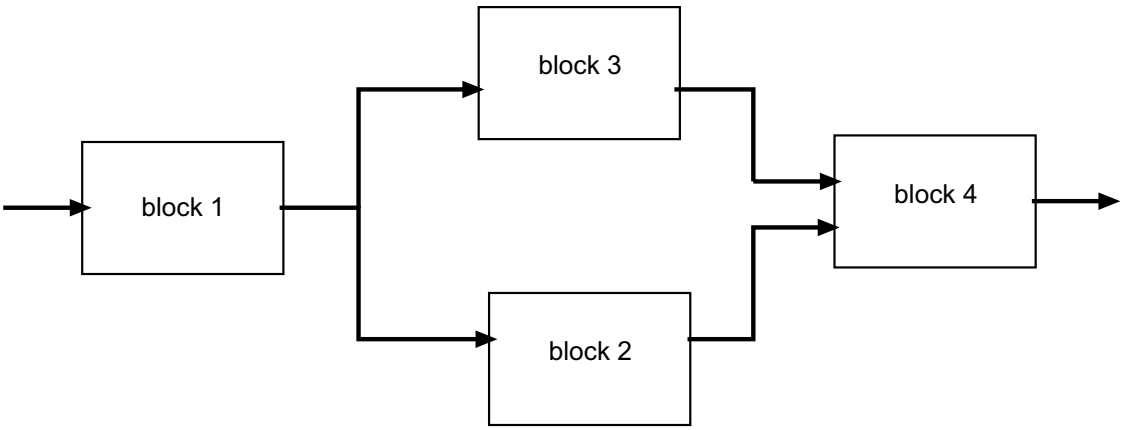
The components listed above are those offered by the prototype version of the AOCS framework. Later version may offer a richer set of default implementations of the framelet interfaces. In particular, they may offer a richer set of default control blocks implementing transfer functions that are useful in AOCS systems.

5 CONTROL CHANNEL CONCEPT

Control channels are the most general type of sequential processing chains. They can represent any sequential multi-input-multi-output processing chain and allow concatenation and nesting of individual processing blocks.

A sequential processing chain is made up of inter-connected processing blocks. More specifically, the term *control channel block* (or simply *block*) will be used to designate a processing block that cannot be further decomposed into lower level processing blocks.

Blocks can be concatenated in chains as shown in the figure:

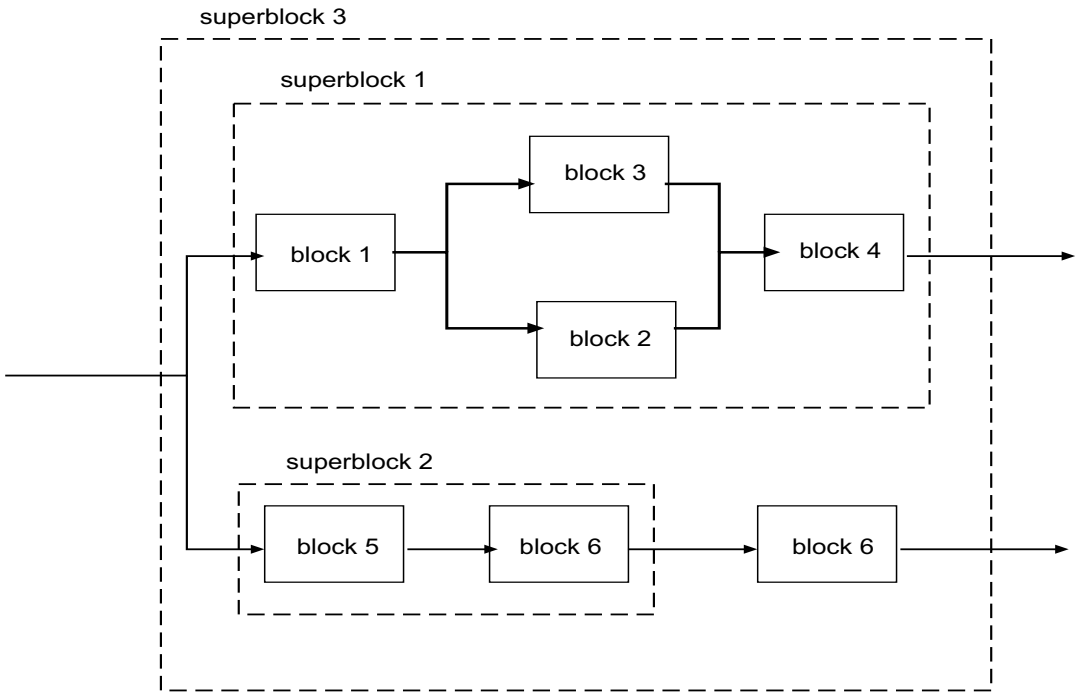


Arrows represent data flow. User input is fed to block 1. User output is taken from block 4.

Blocks chains can be nested within higher-level blocks called *superblocks*. Superblocks and blocks can be freely mixed in processing chains as in the example in the figure in the next page.

The control chain is enclosed in a superblock (superblock 3) with one input and two outputs. The superblocks contains both two superblocks (superblocks 1 and 2) and one simple block.

Note that the terminology of blocks and superblocks is the same as in Xmath. Blocks and superblocks as defined here map to the homonymous concepts in Xmath.



5.1 Data Propagation through Control Channels

In general, a control channel implements a transfer function of the following kind:

$$x_{t+\Delta t} = f(x_t, u_t)$$
$$y_t = g(x_t, u_t)$$

where the usual notation is adopted with u representing the input vector, y the output vector and x the state vector.

The fundamental operation to be performed on a control channel is the propagation of its output signal from time (t-Δt) to time t. A *propagate(t)* operation will be defined on control channel objects that causes their outputs to be propagated up to time t.

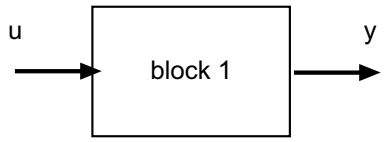
The time t to which the output values are propagated is called the *last propagated time* of the control channel. Thus, at any time a variable `lastPropagationTime` is defined that specifies the time to which state and output were last propagated. A call to `propagate(t)` causes the state and output to be propagated from time `lastPropagatedTime` to time `t`.

In order to compute y(t), the control channel needs to know u(t). Thus, if control channels are arranged in a sequential chain, a propagation request must be passed to upstream blocks.

For example, consider again the control channel chain of the first figure in the previous section. Suppose `propagate(t)` is called on block 4. Before this operation can be performed on this block, it is necessary to update the inputs to the block and this is done by calling the same operation `propagate(t)` on blocks 2 and 3. Thus, propagate requests percolate along the processing chain.

Eventually, they will reach a control channel that takes its inputs from an object that is not itself a control channel. Such external signals will be assumed to be fed to a control channel using a *zero-order hold*. This means that their value will be assumed to be constant across propagation instants.

To illustrate the zero-order hold concept, consider for instance the situation in the figure:



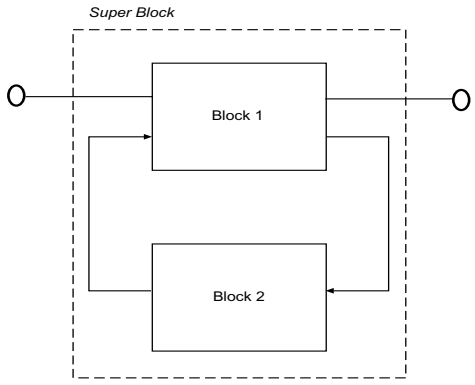
Suppose that both input and output have the same validity time t. Suppose now that operation `propagate(t+Δt)` is called on the control channel. The control channel will responds by updating its output to y(t+Δt) and it will do so by assuming that the input remains constant and equal to u(t) throughout the propagation interval.

If a first-order hold were used, then the value of the input signal would be computed by linear extrapolation from its last two known values. Higher order holding systems are also possible. Zero-order holding is, however, by far the most commonly used type of holding mechanism in satellite control systems and is the only one for which the AOCS framework makes provisions.

5.2 Signal Loops

The mechanism outlined in the previous section cannot cope with signal loops.

Consider the situation shown in the figure:



When the super block receives a `propagate` request, it will route it to block 1. Before executing the `propagate` action, block 1 will try to update its inputs. It will do so by issuing a `propagate` request to block 2. This will in turn try to update *its* inputs and will do so by issuing a `propagate` request to block 1. An endless loop will result.

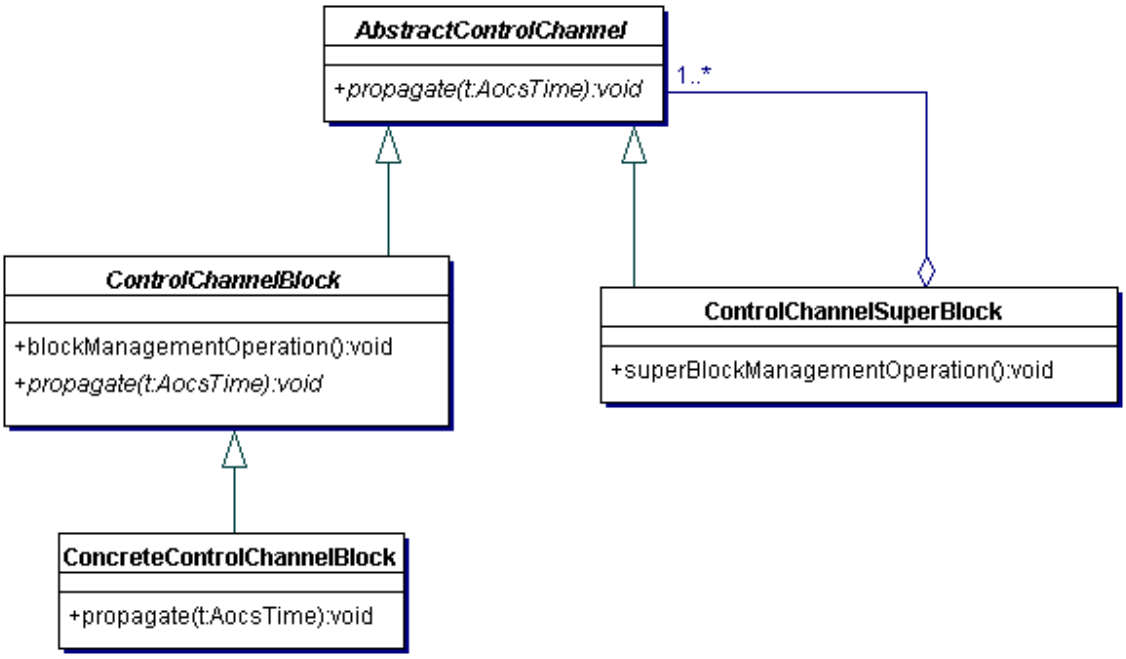
The example shows that signal loops in the control channel connections, either at block or super block level, will result in an endless loop.

The propagation mechanism could be modified to handle loops (at the cost of some overhead) but this is judged unnecessary as signal loops should not arise in an on-board control system.

Note that there is no loop detection mechanism. Responsibility for detecting loops rests with the developer who makes the control channel connections.

5.3 The Control Channel Design Pattern

This design pattern is introduced to model the control channel concept and in particular the distinction between blocks and super blocks. The control channel design pattern is obtained by instantiating the composite pattern and it is shown in the following UML diagram:



`AbstractControlChannel` is a pure interface that exposes the methods that are common to all control channels, regardless of whether they are blocks or superblocks. Control channels are always seen by their clients as instances of this class. Since both blocks and superblocks are derived from `AbstractControlChannel`, they can be treated in a uniform manner as instances of abstract control channels. The fundamental operation exposed by `AbstractControlChannel` is `propagate`(t) that propagates the input signals to the output up to time t as discussed in a previous section. This method gives rise to the control channel hot-spot through which application developers must define the transfer function to be implemented by the control channels they use.

Abstract control channels can be implemented either as control blocks or as control super blocks. `ControlChannelBlock` is an abstract class that acts as base class for all concrete control blocks. It provides concrete implementations of methods and data structure to manage block operations. This class for instance provides data structures to buffer the input and output signals and operations to reset them. Such data structures and operations are common to all control channel blocks. This class is abstract because it does not provide any implementation for method propagate. The implementation of this method defines the concrete transfer function that is implemented by the control channel. Concrete control blocks specialize `ControlChannelBlock` by providing concrete algorithms for the propagation of the input signals through the control block. The AOCS Framework provides as default

components concrete implementations of this class that implement common transfer functions such as PD blocks, PID controllers, integrators, etc.

Super blocks are represented by instances of class `ControlChannelSuperBlock` which is derived from `AbstractControlChannel`. This class defines a default component that manages a set of interconnected lower-level blocks. The lower-level blocks are seen as instances of `AbstractControlChannel` since they can be either control channel blocks or control channel super blocks.

Since both blocks and superblocks are derived from `AbstractControlChannel`, they can be treated in a uniform manner as instances of control channels.

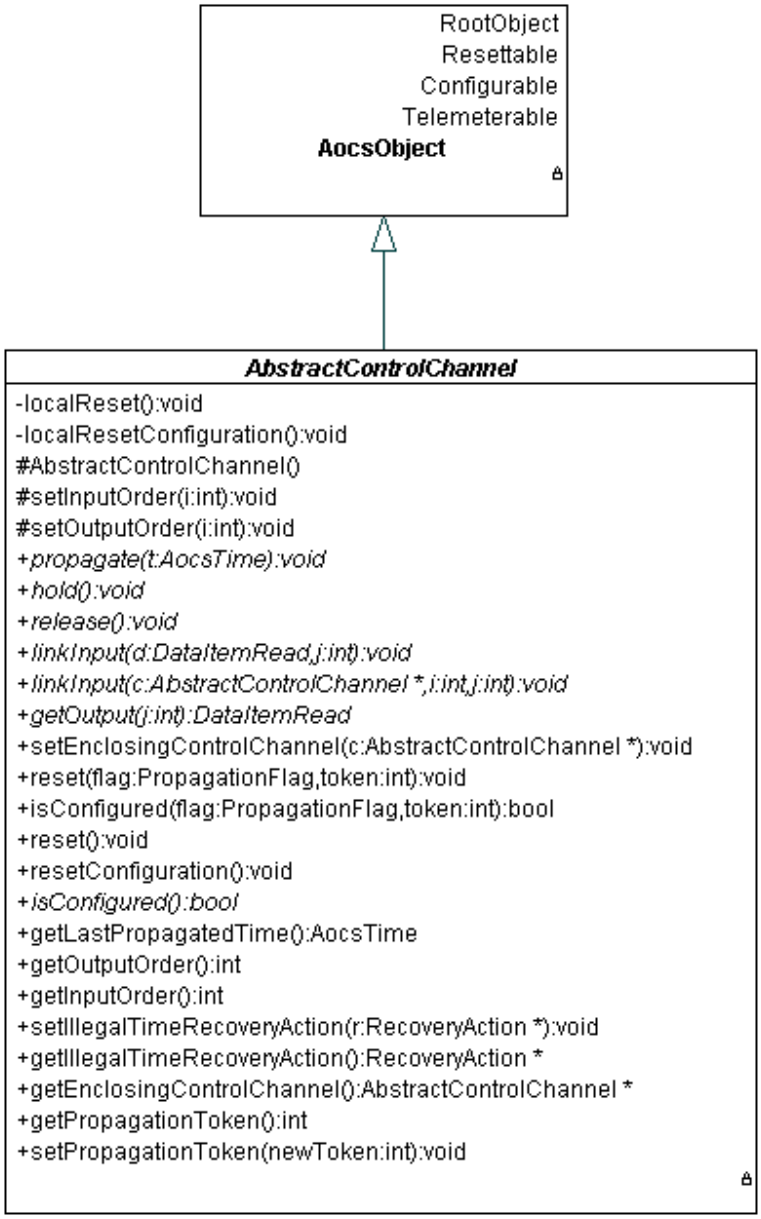
The way this pattern is instantiated in the framework is described in the next sections.

5.4 Recursion

Calls to method `propagate`() can be recursive since when they are called on a given component A, they have to be propagated backward to all control channels directly or indirectly linked to A's inputs. The maximum depth of the recursion is given by the maximum length of a chain of connected control blocks.

6 ABSTRACT CONTROL CHANNELS

The `AbstractControlChannel` class is the interface through which any control channel is seen by its clients:



The semantics of public operations specific to the `AbstractControlChannel` class (ie those not inherited from base classes) are described in the table:

`propagate(t)`
Propagate state and output equations from the current time (as defined by `lastPropagationTime`) to the target time t (see section 5.1).
`hold, release`
See section 6.1.
`linkInput(inputDataItem, i)`
Link the i-th input of the control channel to the data item `inputDataItem` (see section 6.2). Attempts to link a non-existent input will cause a configuration error event to be raised.
`linkInput(&controlChannel, i, j)`
Link the j-th input of the control channel to i-th output of `controlChannel` (see section 6.2). Attempts to operate on a non-existent input or output will cause a configuration error event to be raised.
`getOutput(i)`
Returns the i-th output of the control channel as a `DataItemRead`. Attempts to operate on a non-existent output will cause a configuration error event to be raised.
`getEnclosingControlChannel, getEnclosingControlChannel`
Control channels that are embedded within other control channels, need to have a reference to their enclosing control channel. This reference is maintained in a variable called `enclosingControlChannel` for which these are the getter and setter methods. It is the responsibility of the developer to set the enclosing control channel when the control channels are configured.
`getLastPropagatedTime`

Returns the time to which the state and outout were last propagated (see section 5.1). After a reset, this method returns the time when the control channel was reset.
`getOutputOrder, getInputOrder`
Returns the number of outputs and of inputs in the control channel.
`getIllegalTimeRecoveryAction, setIllegalTimeRecoveryAction`
If method propagate is called with a target time that is lower than `lastPropagatedTime` (see section 5.1), then a failure event is raised. As usual, to this event a recovery action is associated. These methods are the getter and setter methods for such recovery action.
`getPropagationToken, setPropagationToken`
These methods should never be called by the application developers.

Many of the above methods are pure virtual methods since their implementation for control blocks is difference from their implementation for super blocks.

6.1 Hold/Release Operations

State propagation can be temporarily suspended by putting a control channel in *hold mode*. This is effected by calling its `hold` method. When in hold mode, the only component of a control channel's internal state to be updated is its propagation time. This means that the only effect of a to `propagate(t)` when the block is in hold mode is to make `lastPropagatedTime=t`. The control channel state and outputs remain constant.

Normal operation is resumed by calling method `release`.

6.2 Input Linking

The main function of a control channel is to process one or more input signals. The input signals can be taken either from a fixed input source or from the output of another control channels. The latter mechanism allows control channels to be linked in chains.

Linking to an external input source is done through the read data item mechanism. This means that the control channel is given a `ReadDataItem` object that encapsulates a pointer to the input source variable. This type of input link is set up by calling method

`linkInput(inp, i)` which associates the read data item `inp` to the i-th input of the control channel.

Alternatively, the input of a control channel can be linked to the output of another control channel. This is done through method `linkInput(&cc, i, j)` that associates the j-input of the control channel to the i-th output of the source control channel cc.

In linking control channels to each other, care must be taken to avoid signal loops (see section 5.2).

6.3 Control Channel Outputs

The outputs of control channels can only be accessed as instances of type `DataItemRead` by calling `getOutput`.

Any number of control channels can be connected to the same output of a certain control channel. Every time a client of a control channel calls its `getOutput` method, the control channel constructs a new `DataItemRead` giving read-only access to its output.

6.4 The Telemetry Interface

Abstract control channel inherit the `telemeterable` interface. However, they do not provide a class-specific implementation for the associated methods. This is because there is no state information that is associated specifically to abstract control channels and therefore it does not make sense to define class-specific telemetry methods.

6.5 The Reset Interface

Abstract control channels inherit from `AocsObject` the `resettable` interface and must therefore implement the corresponding method.

Method `reset` performs the following actions:

- sets the `lastPropagatedTime` to the current AOCS time.

6.6 The `Configurable` Interface

Abstract control channels are configurable objects and therefore have a non-trivial implementation of the `Configurable` interface which they inherit from their base classes.

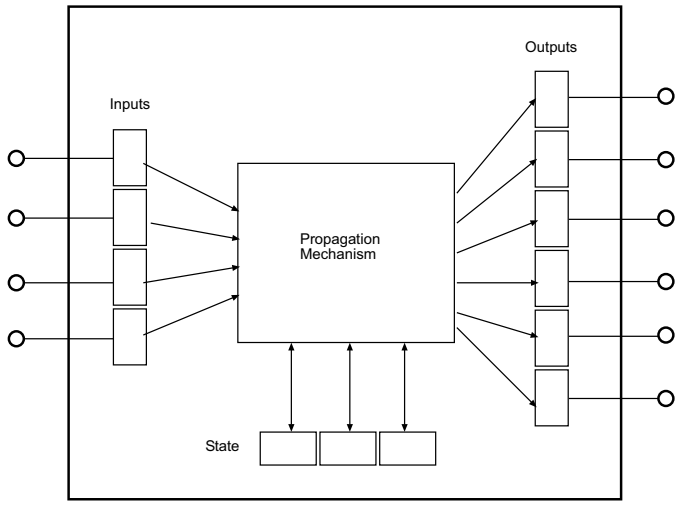
Method `resetConfiguration` performs the following actions:

- resets the embedded control channel pointer to NULL
- resets the pointer to the recovery action associated to the control block to NULL

There is no class-specific implementation of method `isConfigured`.

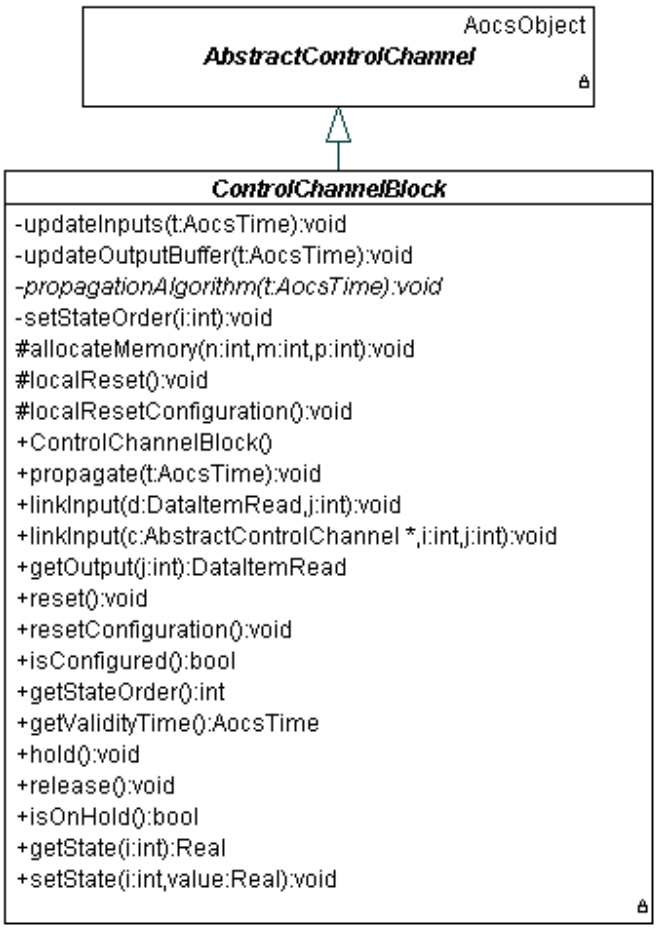
7 CONTROL CHANNEL BLOCK IMPLEMENTATION

The conceptual structure of a control channel block is shown in the figure:



As can be seen in the figure, a control channel consists of buffers holding the input and output signals, a propagation mechanism, and an (optional) set of internal state variables. The dimension of the input and output buffers is given by the number of input and output signals. Given that dynamic memory allocation is not allowed, input and output order will be treated as parameters.

This conceptual structure is encapsulated by the following class:



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

`isOnHold`
Returns true is the control block is currently in hold state (see section 6.1).
`getState(i), setState(i)`
Getter and setter method for the i-th component of the block's internal state. Attempts to operate on a non-existent state will cause a configuration error event to be raised.

`getStateOrder`
Returns the size of the state buffer.

The most significant operations of the control channel class are presented in the next subsections.

7.1 Memory Allocation

Control channel blocks maintain arrays to store the input, output and state buffers. The operations defined at the level of class `ControlChannelBlock` are parameterized by the number of inputs, outputs and states. The concrete number of inputs, outputs and states for a particular control blocks is specified when the *concrete* control block is initialized.

Class `ControlChannelBlock` provides a protected method `allocateMemory` that is used by concrete control blocks to allocate the input, output and state buffers. This method can only be called once. Attempts to call it more than once will cause a configuration error event to be raised.

7.2 Data Propagation

Data propagation is performed by calling `propagate(t)` which will cause the output signals to be updated to the target time t.

The implementation of this method for control blocks is:

```
{
     if (t<lastPropagatedTime)
     {    . . . \\ raise failure event
     }

     updateInputs(t);
     if  ( (t>lastPropagatedTime) || (initFlag) )
     {    if ( !isOnHold() )
          {    propagationAlgorithm(t);
               updateOutputBuffer(t);
          }
          lastPropagatedTime = t;
          initFlag = false;
     }
}
```

The method begins by checking that the target time is *after* the time to which state and output where propagated. If this is not so, a failure event is raised.

As explained in section 5.1, before the propagation takes place, it is necessary to update the inputs. This is done by calling the private operation `updateInputs(t)`. This call essentially causes `propagate(t)` to be called on recursively all upstream blocks.

The `initFlag` is not discussed here as it concerns the implementation of the Xmath interface (see section 7.9).

Actual state and output propagation is done only if the block is not on hold. If it is, the only effect of calling `propagate(t)` is to make `lastPropagatedTime` equal to `t`.

The actual propagation algorithm cannot of course be specified at the level of class `ControlChannelBlock` since this algorithm is specific to each concrete control block. The algorithm is encapsulated in method `propagationAlgorithm` that is the main hot-spot for this class. Concrete classes must provided an implementation for this method to implement the state and output propagation equations.

After the propagation algorithm has been implemented, the output buffers are updated.

7.3 Hold\Resume Operations

Control channel blocks inherit from the abstract control channel interface method `hold` and `resume`. A call to method `hold` essentially freezes the block's state that is no longer updated in response to calls to method `propagate`. When the block is in the hold state, calls to `propagate` simply result in an update of the time and have no effect on either the internal state or the external output.

A call to `resume` updates the validity time to the current time[1] and restores the normal operation of the control block.

In order to understand the possible use of the hold\resume operations consider a control block implementing an integrator. A call to `hold` causes the integrator to stop integrating the input. Note that this is *not* equivalent to not calling `propagate` for the duration of the hold interval.

[1] Note that control blocks have access to the current time through `AocsObject`.

7.4 State Operations

Control blocks normally maintain internal state variables. The number of state variables can be obtained by calling `getStateOrder`. Individual state variables can be read or set as variables of `Real` type with methods `setState` and `getState`. The first argument of these methods specifies the state variables.

7.5 The Telemetry Interface

Control channel blocks inherit the `telemeterable` interface. However, they do not provide a class-specific implementation for the associated methods. The type and format of telemetry information for a control block must be defined at the level of concrete control blocks.

Note that the input and output data to a control channel are also available in other parts of the AOCS software (typically in a data pool) and therefore it does not seem to make sense to send the internal input, state and outputs of a control blocks to telemetry by default.

7.6 The Reset Interface

Control channel blocks inherit from `AocsObject` the `resettable` interface and must therefore implement the corresponding method.

Method `reset` performs the following actions:

- resets the input and output buffers and the internal state of the control channel to zero.

7.7 The `Configurable` Interface

Control channel blocks are configurable objects and therefore have a non-trivial implementation of the `Configurable` interface which they inherit from their base classes.

Method `resetConfiguration` performs the following actions:

- resets all the input links

Method `isConfigured` returns true if all inputs are connected to external data sources.

7.8 Concrete Control Channel Implementation

Concrete control blocks are implemented as subclasses of `ControlChannelBlock.`

Concrete control blocks inherit all the methods of `ControlChannelBlock`. They only need to provide an implementation for the pure virtual method `propagationAlgorithm` to

implement the block-specific propagation algorithm. Normally, they will also provide implementations for the following methods:

- `reset` : class-specific actions to reset the propagation algorithm.
- `resetConfiguration` : class-specific actions to reset the configuration of the propagation algorithm. Typically this implies resetting all the algorithm parameters.
- telemetry methods : handling of class specific telemetry data.

The prototype framework provides the following concrete general purpose control blocks:

- `I_Block`

  Block implementing an integral transfer function. The integration algorithm is:

  $$y(k+1) = y(k) + G * \Delta t * (u(k) + u(k+1)) / 2$$

  G is a settable gain.

- `P_Block`

  Block implementing a proportional transfer function. The implemented algorithm is:

  $$y(k) = G*u(k)$$

  G is a settable gain.

- `D_Block`

  Block implementing a derivative transfer function. The implemented algorithm is:

  $$y(k) = G*(u(k)-u(k-1)) / \Delta t$$

  G is a settable gain. The derivative is saturated not to exceed a settable threshold.

- `DifferenceBlock`

  Block taking the difference of its inputs:

  $$y(k) = u1(k) – u2(k)$$

- `AdderBlock`

  Block adding together its two inputs:

  $$y(k) = u1(k) + u2(k)$$

- `PassThruBlock`

  Block implementing a unitary transfer function.

- `LimitBlock`

  Block saturating its input:

  $$y(k) = u(k) \quad \text{if } u(k)<L \text{ and } u(k)>-L$$

  $$y(k) = L \quad \text{if } u(k)>L$$

  $$y(k) = -L \quad \text{if } u(k)<-L$$

- `SplitterBlock`

  Block splitting its input among n outputs:

  $$y_i(k) = u(k) \quad \text{for } n=1...n$$

- `TwoByTwoMatrixBlock`

  Block implementing 2-by-2 matrix multiplication. The implemented algorithm is:

  $$y(k) = A*u(k)$$

  A is a 2x2 matrix whose elements are settable.

7.9 Interface to Xmath Autocode

The autocode tool of Xmath can generate code implementing an Xmath procedure superblock. The AOCS framework offers a hook where such code can be plugged in. This allows embedding of Xmath super block into framework control channels.

The format of the code generated by the Xmath autocode tool is determined by a *template* file. In the interest of simplicity, the hook in the AOCS framework uses code generated using the default autocode template file.

The autocode template when applied to a procedure block called <`procedureblock`> generates a single C source code file that contains the following four subroutines:

- <`procedureblock`>

  Routine implementing the input/output transfer function implemented by the Xmath procedure superblock

- `init_application_data`

  Routine that initializes any “%variables” used in the Xmath procedure super block

- `subsys_1`

Non-reentrant wrapper around <procedureblock>

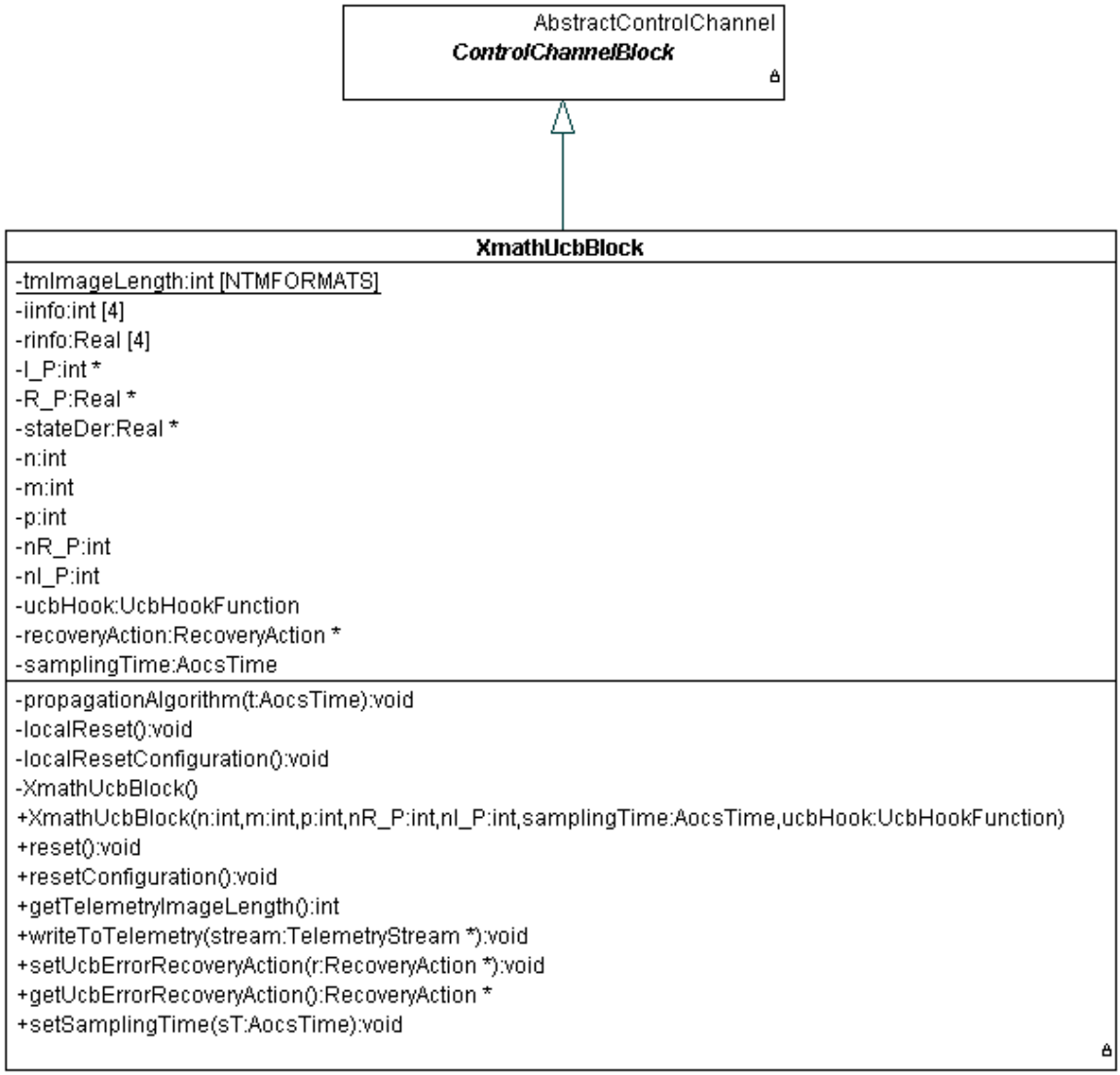
- <procedureblock>_ucbblock

  Reentrant UCB wrapper around <procedureblock>

The AOCS framework offers a component – an instance of class XmathUcbBlock – that acts as a wrapper for the <procedureblock>_ucbblock routine. To be used in the AOCS framework, the component must be initialized by specifying the size of the input, output and state buffers for the procedure superblock and a pointer to routine <procedureblock>_ucbblock. The latter should obviously be linked with the AOCS framework.

Thus embedding of a code from Xmath does not require any manual intervention on the AOCS framework code. All that is needed is the correct initialization of the wrapper component and the linking in of the routine from autocode.

The UML diagram for class XmathUcbBlock is:



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

`XmathUcbBlock(i,j,k,nR,nI,sT,uH)`
Constructor to initialize the component. Its parameters are:

i = number of inputs j = number of outputs k = number of states nR = number of real parameters (should be 0 in the framework) nI = number of integer parameters (should be 0 in the framework) sT = sampling time uH = pointer to the UCB routine generated by the autocode tool
`setUcbErrorRecoveryAction, getUcbErrorRecoveryAction`
The autocode UCB routine maintains an error flag which it uses to report internal errors. The framework wrapper checks this error flag and, if it finds it to be set, it raises a failure event. A recovery action can be associated to this failure event. These are the setter and getter methods for this recovery action.
`setSamplingTime`
Setter method for the block's sampling time. This operation should not be used by AOCS application developers.

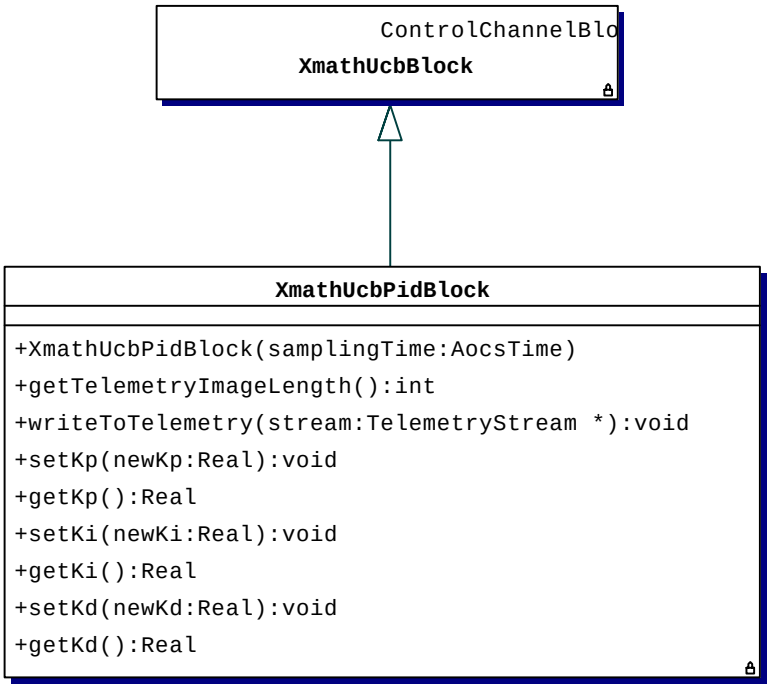
Class `XmathUcbBlock` is derived from class `ControlChannelBlock` and therefore autocode wrapper components can be used like any other control channel with one important restriction. The state and output propagation on control channel blocks can be done at arbitrary times. The semantics of operation `propagate(t)` guarantees that the operation will propagate the block's state and output from its `lastPropagatedTime` to the target time t. There is no requirement that `propagate` be called at regular intervals.

Xmath procedure blocks, however, model discrete transfer functions with fixed sampling times. Their autocode image is therefore designed to be triggered at fixed intervals. Method propagate on `XmathUcbBlock` should therefore also be called at fixed intervals. The interval size (the procedure block sampling time) is one of the initialization parameters for this class.

7.10 Handling of %Variables in Autocode Wrappers

Class `XmathUcbBlock` does not offer any setter and getter methods for the %variables that may be present in the Xmath procedure block. If a procedure super block contains such variables and if it is desired to have access to them from the AOCS framework, it is necessary to derive a class from `XmathUcbBlock` that adds getter and setter methods for the %variables.

The framework prototype offers an example of such a mechanism with class `XmathUcbPidBlock`. This class acts as a wrapper for the code generated from an Xmath PID block. The class diagram for `XmathUcbPidBlock` is:



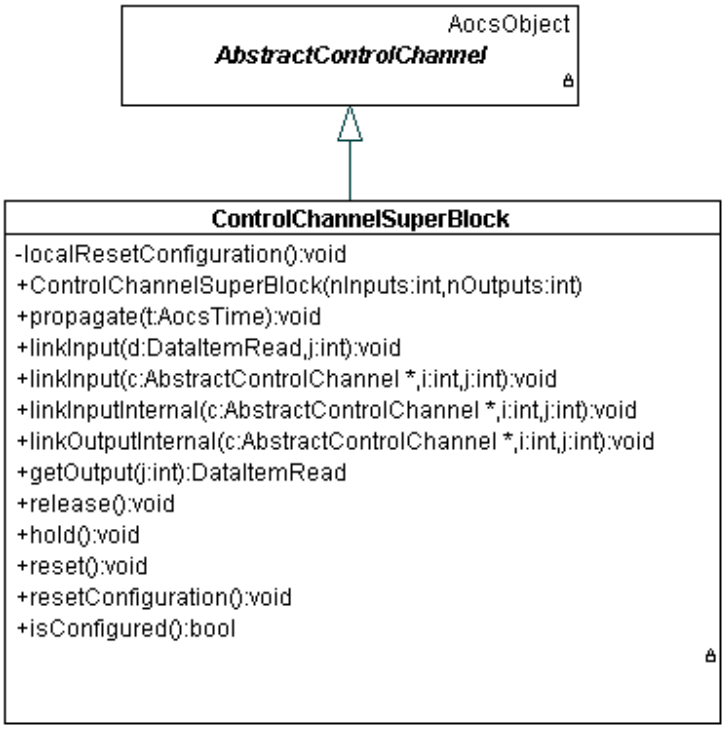
Class `XmathUcbPidBlock` adds to its base getter and setter methods for the three parameters of the PID controller and provides a new implementation of `writeToTelemetry` that sends to the telemetry stream the three PID parameters.

Note that it is not possible to include generic management of %variables in class `XmathUcbBlock` because the %variables have to be accessed by their name and these names are specific to each Xmath procedure block.

It should also be noted that the default values of the %variables are hard-coded in routine `init_application_data`. Hence, whenever a control channel embedding code from an Xmath procedure super block is reset, its %variables are automatically reset to their initial default values. Moreover, because of the way the initialization is performed by the Xmath autocode, these initial default values will always be used the first time the `propagate` operation is called. Changes to the %variables can only be done *after* `propagate` has been called at least once.

8 SUPER BLOCK IMPLEMENTATION

Super blocks are instances of class `ControlSuperBlock`. The UML diagram for this class is:



Since they are indirectly derived from `AbstractControlChannel`, super blocks can also be treated as control channels.

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

`linkInputInternal(&controlChannel, i, j)`
Links the i-th input of `ControlChannel` to the j-th input of the super channel (see section 8.1).
`linkOutputInternal(&controlChannel, i, j)`
Links the i-th output of `ControlChannel` to the j-th output of the super channel (see

section 8.1).

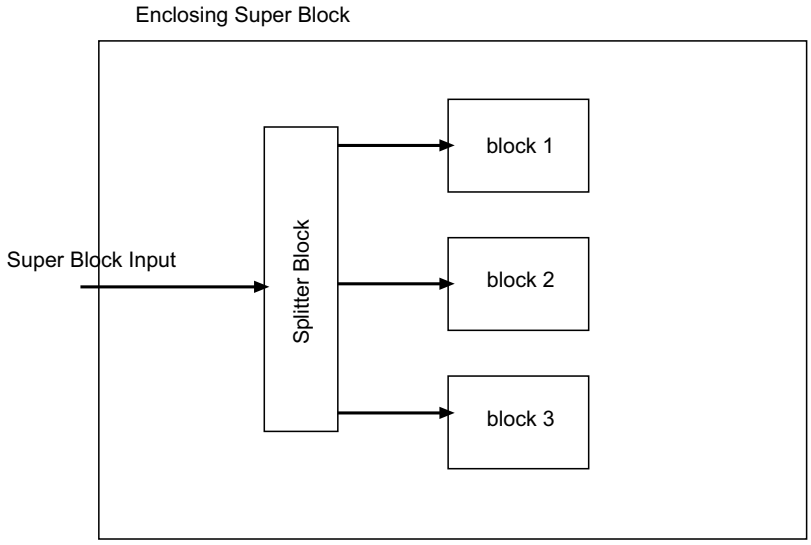
8.1 Embedding Control Channels into Super Blocks

Super blocks act as containers for chains of connected control channels (see figure at the end of section 5). The procedure for embedding control channels into a super block is as follows:

- connect together the control channels that are to be embedded. This is done by calling operation `linkInput` on the control channels and results in a chain of control channels being formed.
- connect the start of the chain to the input of the super block. This is done using operation `linkInputInternal`.
- connect the end of the chain to the output of the super block. This is done using operation `linkOutputInternal`.

Thus in the case of figure at the end of section 5, one would first connect superblock 2 and block 6 and then one would connect the input of the outer superblock to the inputs of superblocks 1 and 2 and the output of the outer superblock to the outputs of superblock 1 and block 6.

Unlike control blocks (see figure at the beginning of section 7), super blocks do not maintain internal buffers to store their inputs and outputs. This means that only one control channel can be internally linked to the same input of the enclosing control channel. If it is desired to connect more than one control channels to the same input, then a splitter block should be used as shown in the figure:



A splitter block has one input and a user-defined number of outputs. Each output is identical to the input. In the configuration in the figure, the splitter block ensures that control channesl `Block_1`, `Block_2` and `Block_3` receive the same copy of the external input of the super block within which they are embedded.

8.2 Data Propagation

Super blocks do not implement any propagation algorithm. They propagate their outputs by propagating the outputs of the control channels they contain.

Thus, the method calls `propagate` on all the control channels connected to the superblock's output. The `propagate` call will then be automatically percolated downstream through the entire control channel chains contained in the super block.

8.3 Hold\Resume Operations

Control channel super blocks inherit from the abstract control channel interface methods `hold` and `resume`. Calls to these methods are directly propagated to all control blocks in the super block. Thus, a call to method `hold` will cause the same method to be called on all blocks in the super block. Calls to `resume` are similarly propagated to the blocks in the super block.

8.4 The Telemetry Interface

The telemetry methods are not implemented in super blocks in the prototype AOCS framework. In a complete implementation, these methods would simply call their counterparts on the blocks contained in the super block.

8.5 The Reset Interface

Super blocks inherit from `AocsData` the `resettable` interface and must therefore implement the corresponding method.

Method `reset` on super blocks does not do anything on the super block itself which has no internal state but it calls the `reset` method on the control channels it contains.

8.6 The `Configurable` Interface

Super blocks are configurable objects and therefore have a non-trivial implementation of the `Configurable` interface that they inherit from `AocsObject`.

Method `resetConfiguration` resets the links to the embedded control block. Essentially, it "empties" the super block. It does not, however, reset the configuration of the enclosed control channels.

Method `isConfigured` returns true if all inputs and outputs are internally connected to blocks inputs and outputs and if all enclosed control channels are configured (ie. if their `isConfigured` methods return `true`).

9 FRAMELET HOT-SPOTS

This section classifies the framelet hot-spots defined in the previous sections of this document. The classification is as described in RD6.

9.1 Control Block Hot-Spot

Name: **Control Block Hot Spot**
Visibility Level: framework-level
Adaptation Time: compile-time
Adaptation Method: derivation from base class `ControlChannelBlock`
Pre-defined Options: control block components exported by the framelet (see section 4)
Related Hot-Spots: none
Description The base class `ControlChannelBlock` implements all the basic mechanisms for a control block but leaves a hook for the algorithm defining the transfer function of the control block. This algorithm is basically implemented by method `propagationAlgorithm`. which is a pure virtual method in `ControlChannelBlock`. Implementation of a concrete control block thus requires derivation of a class from `ControlChannelBlock`. The derived class must, at a minimum, provide an implementation for method `propagationAlgorithm`. Addtionally, it may be necessary to override methods reset, to implement any initialization actions for the propagation algorithm, and the telemetry methds to send some algorithm-specific data to the telemetry stream.

9.2 Recovery Action plug-In for Control Channels

Name: Recovery Action **Plug-In for Control Channels**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `AbstractControlChannel` class (method `setRecoveryAction`)

Pre-defined Options: none. By default, no recovery action is associated to control channels.
Related Hot-Spots: none
Description Control channels raise a failure event when method propagate is called with a target time that is smaller than the current time. This hot-spot allows a recovery action to be associated to this failure.

9.3 Data Input link for Control Channels

Name: Data Input link **for Control Channels**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in object in `AbstractControlChannel` class (method `linkInput`)
Pre-defined Options: none
Related Hot-Spots: control channel input link for control channels
Description Control channels can be set up to take their input from an input source defined by a `DataItemRead` object. The link is set by calling method `linkInput`.

9.4 Control Channel Input link for Control Channels

Name: Control Channel Input link **for Control Channels**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `AbstractControlChannel` class (method `linkInput`)
Pre-defined Options: none
Related Hot-Spots: data input link for control channels

Description Control channels can be set up to take their input from the output of another control channel. The link with the input control channel is done by calling method `linkInput`.

9.5 Embedding of Control Channels in Super Blocks

Name: Embedding of Control Channels in Super Blocks
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in components in `ControlChannelSuperBlock` class
Pre-defined Options: none
Related Hot-Spots: none
Description Super blocks act as containers for other control channels. The lower level control channels are embedded in a super block by using methods `linkInputInternal` and `linkOuputInternal`.

9.6 Hold/Resume Hot Spot

Name: Hold/Resume Hot Spot
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: feature enable/disable (methods `release` and `resume`)
Pre-defined Options: none. By default control channels are in *released state*.
Related Hot-Spots: none
Description State propagation in control channels can be dynamically put on hold by calling the `hold` method and can be dynamically resumed by calling the `resume` method.

9.7 Xmath UCB Autocode Hot-Spot

Name: Xmath UCB Autocode Hot-Spot
Visibility Level: framework-level
Adaptation Time: compile-time
Adaptation Method: parameter in constructor for `XmathUcbBlock` class
Pre-defined Options: none.
Related Hot-Spots: none
Description Class XmathUcbBlock acts as a wrapper for an UCB autocode routine from the Xmath. The UCB routine is inserted into the AOCS framework by passing a pointer to it as a parameter to the constructor for class `XmathUcbBlock`. The UCB routine must then be linked with the AOCS framework.

9.8 UCB Error Recovery Action Plug-In

Name: UCB Error Recovery Action **Plug-In**
Visibility Level: framework-level
Adaptation Time: run-time
Adaptation Method: plug-in component in `XmathUcbBlock` class (method `setRecoveryAction`)
Pre-defined Options: none. By default, no recovery action is associated to UCB errors.
Related Hot-Spots: none
Description Class `XmathUcbBlock` acts as a wrapper for an UCB autocode routine from the Xmath. The UCB routine maintains a flag with which it reports internal errors. The framework wrapper checks this error flag and, if it finds it to be set, it raises a failure event. A recovery action can be associated to this failure event.

10 FRAMELET FUNCTIONALITIES

This section defines the functionalities offered by the framelets together with their mutual relationships and their mappings to framelet architectural constructs. The definition follows the guidelines of RD6.

10.1 Conventions

The functionality code defines the type of the functionality according to the following convention:

- CF = can-functionality
- DF = do-functionality
- OF = offer-functionality

The following numbering conventions are used:

- if Fx is a functionality, then the functionalities that are obtained by expanding it are numbered as Fx.n where n is 1, 2, 3, etc
- if CFx is a can-functionality, then the offer-functionality that implement it are numbered CFx,n where n is 1, 2, 3, etc

10.2 Functionality List

The functionalities for the sequential data processing framelet are shown in the table below. Each entry covers one functionality giving its definition, its relationships to other functionalities (if any) and its mappings to framelets architectural constructs (if any).

CF1	The sequential data processing framelet can implement any control channel
	expands to *DF1.1 and CF1.2*
DF1.1	The sequential data processing framelet provides a generic and customizable control channel super block
	is-implemented-by *ControlChannelSuperBlock component*

CF1.2	The sequential data processing framelet can implement control channel blocks implementing an arbitrary transfer function
	matches *Control Block Hot-Spot*
OF1.2.1	The sequential data processing framelet offers a control block implementing a proportional transfer function.
	is-implemented-by *P_Block Component* ***matches*** *Control Block Hot-Spot*
OF1.2.2	The sequential data processing framelet offers a control block implementing a derivative transfer function.
	is-implemented-by *D_Block Component* ***matches*** *Control Block Hot-Spot*
OF1.2.3	The sequential data processing framelet offers a control block implementing an integral transfer function.
	is-implemented-by *I_Block Component* ***matches*** *Control Block Hot-Spot*
OF1.2.4	The sequential data processing framelet offers a control block implementing a difference function.
	is-implemented-by *DifferenceBlock Component* ***matches*** *Control Block Hot-Spot*
OF1.2.5	The sequential data processing framelet offers a control block implementing a unitary transfer function.
	is-implemented-by *PassThruBlock Component* ***matches*** *Control Block Hot-Spot*
OF1.2.6	The sequential data processing framelet offers a control block implementing a saturation transfer function.

	is-implemented-by *LimitBlock Component* ***matches*** *Control Block Hot-Spot*
OF1.2.7	The sequential data processing framelet offers a control block implementing an adder transfer function.
	is-implemented-by *AdderBlock Component* ***matches*** *Control Block Hot-Spot*
OF1.2.8	The sequential data processing framelet offers a control block implementing a splitter transfer function that split an input among n identical outputs.
	is-implemented-by *SplitterBlock Component* ***matches*** *Control Block Hot-Spot*
OF1.2.9	The sequential data processing framelet offers a control block implementing a 2x2 matrix multiplication transfer function.
	is-implemented-by *TwoByTwoMatrixBlock Component* ***matches*** *Control Block Hot-Spot*
OF1.2.10	The sequential data processing framelet offers a control block implementing a PID controller.
	is-implemented-by *XmathUcbPidBlock Component* ***matches*** *Control Block Hot-Spot*
OF1.2.11	The sequential data processing framelet offers a control block that embeds a generic UCB routine generated by the Xmath autocode tool.
	is-implemented-by *XmathUcbBlock Component* ***matches*** *Xmath UCB Autocode Hot-Spot*
DF1.2.6.1	The Xmath control block of OF1.2.6 reports failures detected by the UCB routine by the Xmath autocode tool.

	is-implemented-by *XmathUcbBlock Component* ***uses*** *DFx from component communication framelet (failure reporting service)*
CF1.2.6.2	Any recovery action can be associated to the failure report of DF1.2.6.1.
	matches *the UCB Error Recovery Action Plug-In Hot Spot* ***uses*** *DFx from component communication framelet (association of recovery actions to failure events)*
DF1.1.1	Control channels are telemeterable.
	is-implemented-by *ControlChannelSuperBlock and ControlChannel Block components* ***uses*** *CF3 from telemetry framelet*
DF1.1.2	Control channels provides reset and configuration services.
	is-implemented-by *ControlChannelSuperBlock and ControlChannel components* ***uses*** *CF1 from the system management framelet (reset services)* ***uses*** *CF2 from the system management framelet (configuration services)*
CF1.1.3	Any non-cyclical chain of linked control channels can be embedded in a control channel super block.
	matches *Embedding of Control Channels in Super Blocks Hot-Spot*
CF2	State propagation inside any control channel can be held or resumed dynamically.
	matches *the hold/resume Hot-Spot*
CF3	The input of a control channel can be dynamically linked to any data item object.
	matches *the data input link Hot-Spot* ***uses*** *DFx from component communication framelet (data item components)*

CF4	The input of a control channel can be dynamically linked to the output of another control channel provided that the linking operation does not give rise to any cycles.
	matches *the control channel input link Hot-Spot*
DF5	Control channels check for and report configuration errors occurring during the input and output linking process.
	is-implemented-by *ControlChannelBlock and ControlChannelSuperBlock components* ***uses*** *DFx from component communication framelet (configuration error reporting service)*
DF6	Control channels check that the target propagation time is greater than the current time and report a failure if this is not the case.
	is-implemented-by *ControlChannelBlock component* ***uses*** *DFx from component communication framelet (failure reporting service)*
DF7	Any recovery action can be associated to the failure report of DF6.
	matches *Recovery Action Plug-In for Control Channels.* ***uses*** *DFx from component communication framelet (association of recovery actions to failure events)*