THE CORDET FRAMEWORKS
Domain Design

Prepared by P&P Software GmbH
for the Study on Component Oriented Development Techniques
(ESA-Estec Contract 20463/06/NL/JD)

Written By:	A. Pasetti O. Rohlik
Date:	12 September 2008
Issue:	1.2
Reference:	PP-FW-COR-0002

Table of Contents

1 GLOSSARY AND ACRONYMS

The table defines the most important technical terms and abbreviations used in this document.

Term	Short Definition
Abstract Interface	A definition of the signature and semantics of a set of logically related operations without any implementation details.
AOCS	The Attitude and Orbit Control Subsystem of satellites.
Control Framework	A framework covering the AOCS subsystem (one of the two frameworks presented in this document).
Application	A software program that can be deployed and run as a single executable.
Application Instantiation	The process whereby a component-based application is constructed by configuring and linking individual components.
Component	A unit of binary reuse that exposes one or more interfaces and that is seen by its clients only in terms of these interfaces.
Component-Based Framework	A software framework that has components as its building blocks.
Computational Node	A computational resource that has memory and processing capabilities.
CORBA	A widely used middleware infrastructure.
Design Pattern	A description of an abstract design solution for a common .
DH	Data Handling (one of the functional subsystems of an on-board system).
DH Framework	A framework covering the DH subsystem (one of the two frameworks presented in this document).
Domain	A short-hand for either 'family domain' or 'framework domain'.
DSL	Domain Specific Language (a language that is created to describe applications or components in a very narrow domain).
DTD	Document Type Definition. It defines the legal building blocks of an XML document. It defines the document structure with a list of legal elements. Its purpose is similar to the one of an XML Schema, although it is not as feature rich and the syntax is different.
EMF	Eclipse Modelling Framework: a modeling framework and code generation facility for building tools and other applications based on a structured data model.
Family Domain	The set of systems whose implementation is supported by a system or product family.
FDIR	Failure Detection Isolation and Recovery.
Feature	A characteristics of a system or an application that is relevant to its users.
Feature Model	A description of a set of features and their legal combinations.
Framework Domain	The set of applications whose implementation is supported by the framework.
Framework Instantiation	The process whereby a framework is adapted to the needs of a specific application within its domain.
Functional Property	A property that can be expressed as a logical relationship among the variables that define the state of an application or system.
Generative Programming	A software engineering paradigm that promotes the automatic generation of an implementation from a set of specifications.
Generic Architecture	A set of reusable and adaptable software assets to support the instantiation of systems within a certain target domain. In the CORDET project, a generic architecture consists of a system family, to model the non-functional aspects of systems in the architecture's target domain, and a set of software frameworks, to model their functional aspects. The objective of the CORDET Project is to define a generic architecture for satellite on-board systems.
GNC	Guidance Navigation and Control (a synonym for *AOCS*).
Interface	An abstract specification of services to be provided by any concrete realisation of it.
JVM	Java Virtual Machine.
Non-Functional Property	A property other than a functional property.

Object Oriented Framework	A software framework that uses inheritance and object composition as its chief adaptation mechanisms.
OBS	The On-Board Software.
OBS Framework	A prototype framework for on-board systems developed by P&P Software (see [RD18]).
OtM Adaptability	Outside-the-Model Adaptability. An adaptability mechanism that is defined outside the UML2 model.
Product Family	A set of applications or systems that can be built from a pool of shared assets.
Property	Same as a 'feature' above.
Software Component	A unit of binary reuse that exposes one or more interfaces and that is seen by its clients only in terms of these interfaces.
Software Framework	A kind of product family where the shared assets are software components embedded within an architecture optimized for a certain domain and the 'product' is a software application.
System	A group of independent but interrelated hardware and software elements comprising a unified whole.
System Family	A kind of product family where the 'product' to be built using the reusable assets provided by the family is the architectural infrastructure (the 'middleware') of a complex system.
XML	Extensible Markup Language. XML documents consist (mainly) of text and tags, and the tags imply a tree structure upon the document. An XML document is said to be valid if it conforms to an XML Schema or a DTD.
XML Schema	The XML Schema language is also referred to as XML Schema Definition (XSD). They provide a means for defining the structure, contents and semantics of XML documents. XML Schemas are written in XML
WtM Adaptability	Within-the-Model Adaptability Mechanism. An adaptability mechanism that is defined within the UML2 model.

2 REFERENCES

RD1 AOCS Framework Project Web Site, control.ee.ethz.ch/~ceg/AocsFrameworkProject

RD2 RT Java AOCS Framework Project Web Site, control.ee.ethz.ch/~ceg/RealTimeJavaFramework

RD3 Brauer (1999) Object Oriented Languages Study. Final Report for ESA contract 12889/NL/PA

RD4 Clemens P, Northrop L (2001) *A Framework for Software Product Line Practice,* Software Engineering Institute, Carnegie Mellon University, Available from Internet Web Site: www.sei.cmu.edu/activities/plp/framework.html

RD5 Donohoe P (ed), *Software Product Lines – Experience and Research Directions,* Kluwer Academic Publisher

RD6 Fayad M, Schmidt D, R. Johnson R (eds), *Building Application Frameworks – Object Oriented Foundations of Framework Design,* Wiley Computer Publishing, 1999

RD7 Gamma E, et al, *Design Patterns – Elements of Reusable Object Oriented Software,* Addison-Wesley, Reading, Massachusetts

RD8 TimeSys Home Page, http://www.timesys.com/index.cfm

RD9 AERO Project Home Page, http://www.aero-project.org

RD10 Aicas Home Page, http://www.aicas.com

RD11 OBOSS Home Page, http://spd-web.terma.com/Projects/OBOSS/Home_Page/

RD12 Pasetti A, et al, *An Object-Oriented Component-Based Framework for On-Board Software*, Proceedings of the Data Systems In Aerospace Conference, Nice, May 2001

RD13 Pasetti A., *Software Frameworks and Embedded Control Systems*, LNCS Series, Springer-Verlag, 2001

RD14 Szyperski C, *Component Software*. Addison Wesley Longman Limited, Harrow (UK), 1998

RD15 Czarnecki, K., Eisenecker, U.: *Generative Programming – Methods, Tools, and Applications,* Addison-Wesley, 2000

RD16 Birrer I, Chevalley P, Pasetti A, Rohlik O, *An Aspect Weaver for Qualifiable Applications*, Proceedings of the Data Systems in Aerospace (DASIA) Conference, Nice 2004.

RD17 XWeaver Web Site: http://www.pnp-software.com/XWeaver

RD18 OBS Framework Web Site, http://www.pnp-software.com/ObsFramework

RD19 Introduction to Aspect Oriented Programming, http://www.pnp-software.com/AspectOrientedProgramming.html

RD20 Birrer I, Pasetti A, Rohlik O, *Implementing Adaptability in Embedded Software through Aspect Programs,* IEEE Proceedings of the Mechantronic & Robotics Conference 2004, Aachen, Germany, Sept. 2004

RD21 Automated Framework Instantiation Project Web Site, http://www.pnp-software.com/AutomatedFrameworkInstantiation

RD22 Cechticky V, Pasetti A, Rohlik O, Schaufelberger W, *XML-Based Feature Modelling,* published in: J. Bosch, C. Kueger (eds), *Software Reuse: Methods,* *Techniques, and Tools,* LNCS Vol 3107, Springer-Verlag, 2004

RD23 Cechticky V, Chevalley P, Pasetti A, Schaufelberger W, *A Generative Approach to Framework Instantiation,* published in: F. Pfenning, Y. Smaragdakis (eds), *Generative Programming and Component Engineering,* LNCS Vol 2830, Springer-Verlag, 2003

RD24 ASSERT Project – HRI pilot project, Deliverable D6.3.1-1 : *HRI System Family Definition Report*

RD25 ASSERT Project – HRI pilot project, Deliverable D6.3.2-1 : *HRI Reference Architecture Definition Report V1*

RD26 C. Moreno, G. Garcia, *Plug & Play architecture for on-board software components*, Proceedings of the DASIA 2002 conference, Nice 2002

RD27 ASSERT Project – ETH Deliverable D4.2.2-1 : *Software Building Blocks Adaptation Techniques – The FW Profile*

RD28 ASSERT Project – ETH Deliverable D4.2.4-3 : *Contribution to V2 Demonstrator*

RD29 ASSERT Project – ETH Deliverable D4.2.4-4.1 : *The ETH Demonstrator Framework – Contribution to V3 Demonstrator, Part 1 (Domain Design)*

RD30 ASSERT Project – ETH Deliverable D4.2.4-4.2 : *The ETH Demonstrator Framework – Contribution to V3 Demonstrator, Part 2 (Domain Implementation)*

RD31 ASSERT Project – UPD Deliverable D3.1.1-3 : *Catalogue of VM-level Containers*

RD32 Egli M, Pasetti A, Rohlik O, Vardanega T, *A UML2 Profile for Reusable and Verifiable Real-Time Components*, in: Morisio M (ed), Reuse of Off-The-Shelf Components, LNCS Vol 4039, Springer-Verlag, 2006

RD33 ASSERT Project – ETH Deliverable D4.2.3-1 : *Product Family Meta-Model Definition – A Family Meta-Model for the XFeature Tool*

RD34 GMV, *Standard, Methods, and Tools Review for Domain Analysis and Domain Design Execution,* CORDET Deliverable GMV-CORDET-WP202-RP-01

RD35 P&P Software GmbH, *The CORDET Methodology*, CORDET Deliverable Document PP-MR-COR-001, http://www.pnp-software.com/cordet

RD36 ECSS-E70-41A, *Ground System and Operation – Telecommand and Telemetry Packet Utilization*, 30th January 2003

RD37 P&P Software GmbH, *The CORDET Frameworks – Domain Analysis*, CORDET Deliverable Document PP-FW-COR-001, http://www.pnp-software.com/cordet

RD38 FW Profile Home Page, http://www.pnp-software.com/fwprofile

3 INTRODUCTION

This document is written as part of the ESA study on *Component Oriented Development Techniques* or CORDET. The study is done under ESA contract 20463/06/NL/JD.

3.1 Objectives Of The CORDET Study

The general objective of the CORDET study is the definition of a *generic architecture* for on-board satellite applications.

The term “generic architecture” is used to designate a set of reusable and adaptable software assets to support the instantiation of systems within a certain target domain. In the CORDET project, a generic architecture consists of a *system family*, to model the non-functional aspects of systems in the architecture's target domain, and a set of *software frameworks*, to model their functional aspects.

The terms “system family” and “software frameworks” are used to designate two kinds of *product families*. A product family is a set of applications or systems that can be built from a pool of shared assets. A system family is a kind of product family where the 'product' to be built using the reusable assets provided by the family is the architectural infrastructure (the 'middleware') of a complex system. A software framework is a kind of product family where the 'product' to be built is a software application and the shared assets are software components embedded within an architecture optimized for a certain domain. The framework thus provides the building blocks for the development of an application.

The generic architecture to be defined in this study is called the *CORDET Generic Architecture*. The product families which constitute the CORDET Generic Architecture are called the *CORDET Product Families*.

Against this background, the more specific objectives of the CORDET study are:

- To define a methodology for the development of the CORDET Generic Architecture and, by implication, for product family-based development activities at both system- and software-level for satellite on-board applications.
- To identify and to define at the level of their functional and non-functional interfaces the product families that constitute the CORDET Generic Architecture.
- To demonstrate the proposed methodology and the proposed architecture by instantiating a subset of its product families to build an end-to-end demonstrator of an on-board system.
- To get feedback from the space community in order to reach as large an agreement as possible on the outputs of the CORDET study.

The CORDET Methodology, covering the first of the four objectives listed above, is defined in RD35. Familiarity with reference document RD35 is a pre-requisite for an appreciation of the present document.

3.2 Objective Of This Document

The CORDET Methodology foresees that the CORDET Generic Architecture be split into a functional and non-functional part and that each part be developed in three phases (domain analysis, domain design, and domain implementation).

The present document covers the *domain design phase* for the *functional part* of the CORDET Generic Architecture. The functional part of the CORDET Generic Architecture consists of a set of *software frameworks*, one for each functional subsystem of an on-board system. This document is concerned with the *DH Framework*, which covers the Data Handling (DH) Subsystem, and with the *Control Framework*, which covers the Attitude and Orbit Control (AOCS) Subsystem.

The results of the *domain analysis phase* for the CORDET Frameworks are presented in RD-37. Familiarity with reference document RD-37 is a recommended pre-requisite for an understanding of the present document.

The output of the domain design phase is a *design model*. This document presents the design models for the DH and Control Frameworks of the CORDET Generic Architecture.

3.3 Design Model Structure

The CORDET Methodology in RD-35 identifies five possible forms of family-level design models. The CORDET Frameworks use two of these five forms: the *single-model form*, and the *meta-model form*.

The design model of the DH Framework takes the form of a single UML2 model. This model covers all the functional properties and factors of variations specified in the domain model of the DH Framework in RD-37.

The design model of the Control Framework is split into two parts. The first part consists of a single UML2 model. This model covers the functional properties and factors of variations specified for the activity and mode management concepts of the domain model of the Control Framework in RD-37.

The second part of the of design model of the Control Framework consists of two meta-models which cover the functional properties and factors of variations specified for the parameter database and data pool concepts of the domain model of the Control Framework in RD-37.

In order to promote the interoperability of the Control and DH Frameworks (see requirement MR7.3-2 in RD-35), the two UML2 models are packaged and distributed as one single model.

The resulting structure of the design model of the CORDET Frameworks is illustrated in figure 3.3-1.

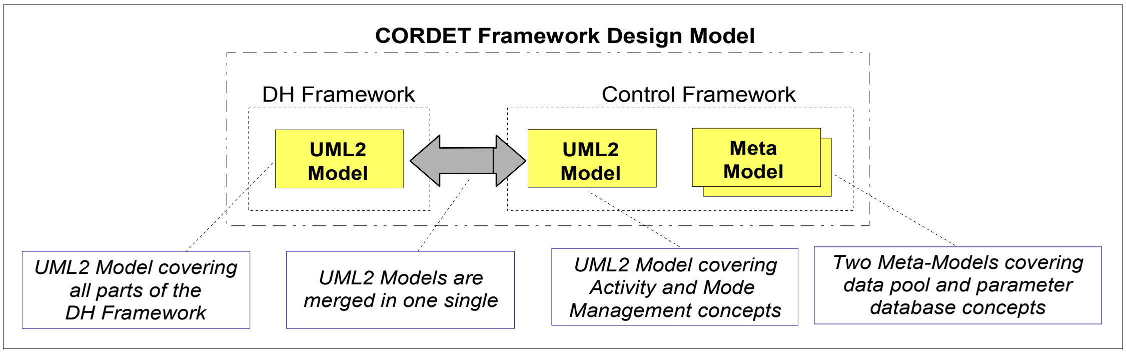

CORDET Framework Design Model
DH Framework
Control Framework
UML2 Model
UML2 Model
Meta Model
UML2 Model covering all parts of the DH Framework
UML2 Models are merged in one single
UML2 Model covering Activity and Mode Management concepts
Two Meta-Models covering data pool and parameter database concepts

Fig. 3.3-1: Structure of CORDET Framework Design Model

3.4 Design Files

This document only gives an overview of the design model of the DH and Control Frameworks. The design model of the two frameworks is formally defined in a set of *design files*.

The CORDET Frameworks design files can be accessed from the "Domain Design" page of the CORDET Web Site[1]. The design files contain their own documentation at the detailed design level. The purpose of the present document is therefore to give a high-level overview of the CORDET Frameworks Design Model to put readers in a position to understand and use the detailed design information in the design files.

The CORDET web site also specifies the licence terms under which the framework design models are made available.

3.5 Terminology

The term "Control Framework" is used in this document to designate the framework that covers the AOCS subsystem. This name was selected because, although the framework was initially targeted at the AOCS, it would also more generally be suitable for on-board control systems.

The term "DH Framework" is used in this document to designate the framework that covers the DH subsystem. More specifically, the DH Framework covers the telecommand and telemetry management within a PUS application.

The two frameworks are intended to be independent of each other. Note, however, that methodological requirement MR7.3-2 in RD-35 requires the two frameworks to be interoperable in the sense that the reusable assets that they provide should be designed to be deployable within the same application.

Interoperability is achieved by having the framework components adhere to the same general design rules; by deriving them from the same basic components; and by packaging their UML2 model into one single model.

3.6 Structure Of This Document

The next section describes the general design rules that have been adopted for the UML2 models of the two frameworks.

The following eight sections describe the UML2 models of the two frameworks. Sections 5 to 8 cover the DH Framework and sections 9 to 13 cover the Control Framework. For each framework, a section is provided that describes the framework architecture and two additional sections are provided that describe how the functional properties and factors of variations defined at domain analysis level for each framework are mapped to design constructs in the UML2 models.

Finally, section 14 covers the meta-model parts of the Control Framework.

[1] http://www.pnp-software.com/cordet

4 GENERAL DESIGN ISSUES

This section covers general design issues that are common to both the DH and the Control parts of the CORDET Frameworks Design Model.

4.1 Naming Conventions

Table 4.1-1 lists the conventions that have been used in naming the elements of the CORDET Frameworks Design Model.

Naming conventions are useful because they increase the readability of the model, and because they can give clues to the code generator that transforms the UML2 model into source code.

Table 4.1-1: Naming Conventions adopted in the CORDET Frameworks Design Model

Element	Description
Classes	The first letters of class names is capitalized.
Interfaces	The names of interfaces have the form: `I<name>` where `<name>` is the name of the topmost class that implements interface `I<name>`. For every class there must be an implementing class defined in the framework even if the class is to be abstract.[2]
Getter Methods	The names of getter methods (methods that return the value of a non-boolean attribute) have the form: `get<name>` where `<name>` is the name of the attribute.
Setter Methods	The names of setter methods (methods that set the value of an attribute) have the form: `set<name>` where `<name>` is the name of the attribute.
Boolean Query Methods	The names of boolean query methods (methods that return the value of a boolean attribute) have the form: `is<name>` where `<name>` is the name of the attribute.
State Query Methods	The names of state query methods (methods that check whether a state machine is in a certain state) have the form: `isState<name>` where `<name>` is the name of the state.
State Machines	The names of state machines have the form: `SM_<name>` where `<name>` is the name of the class.
State Machine States	The names of the states of state machines are written in capitals. The names of state machine states are unique within the framework i.e. no two states – even in different state machines – have the same name. Exception to this rule are the states being extended in which case the state being extended must bear the same name as the corresponding state in the base state machine.
Primitive Types	The names of primitive types that are defined in the model (as opposed to being imported from the standard UML primitive type package) have the form: `TD<name>` where `<name>` is the name of the primitive type. One exception to this rule is the `void` type.
Call Event	Call events are linked to trigger operations. Where there is no danger of ambiguities, their name is the same as the name of the trigger operation to which they are associated. Where there is a possibility of ambiguity (namely where two call events are associated to two operations in the same package with the same name), then their names have the form: `<class>.<operation>` where

[2] UML to RCM gateway requires this.

Element	Description
	`<class>` is the name of the class where the trigger operation is defined and `<operation>` is the name of the trigger operation.
State Transitions	For other state transitions the following naming convention is recommended (but not required). The names of state transitions that originate from the initial or history pseudo-states have the same name: `t0`. The names of a state transition from state S1 to state S2 has the name `S1_S2`. If more than one state transition from S1 to S2 is present, then their names are characterized by a suffix of the form `_i` where 'i' is the transition number (1, 2, 3, etc). Transitions to and from choice pseudo-states follow the same naming convention since the choice pseudo-states can also bear a name.
Entry & Exit Actions	Entry and Exit actions have no name.

4.2 Classes and Interfaces

Figure 4.2-1 illustrates the general class structure of the CORDET Frameworks Design Model. For each class, an interface is defined that declares the public methods of the class. The name of the interface adheres to the naming conventions stated in table 4.1-1.

The inheritance structure of the classes is duplicated at interface level. In other words, if `Class2` is derived from `Class1`, then interface `IClass2` (the interface implemented by class `Class2`) extends interface `IClass1` (the interface implemented by class `Class1`).

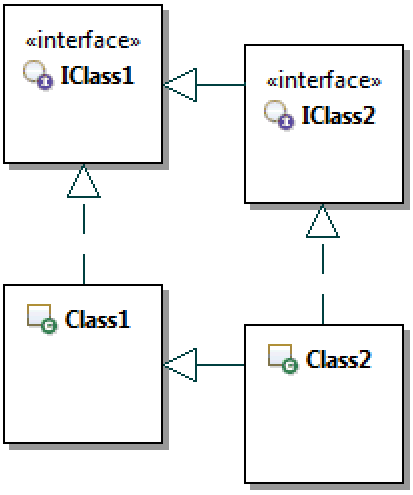


Fig. 4.2-1: Classes and Interfaces

Classes are coupled to each other exclusively through interfaces. In other words, methods never refer to other classes directly but only through their interfaces. This increases the decoupling between concrete or abstract classes. At implementation level, however, use of interfaces may introduced undesirable overheads. In such cases, the adoption of the naming convention for the interfaces makes it easy to build a code generator that “by-passes” interfaces and couples classes directly to each other (in this cases, interface would no longer exist at implementation level).

Note that the use of the interfaces is, in a sense, superfluous. It is retained because it is standard practice in object-oriented designs and because it facilitates a more natural coupling of functional components to their non-functional containers.

4.3 Primitive Types

All the primitive types used in the CORDET Frameworks Design Model have been defined *ex novo*. The standard UML primitive types were not used due to uncertainties about their semantics. According to FW Profile [RD38], primitive data types of integer or boolean type are stereotyped with either `FwInteger` or `FwBoolean` stereotype.

All the primitive types defined for the CORDET project are gathered together in a single package called `com.pnp-software.cordet.primitivetypes`.

4.4 Mapping to UML2 Elements

This section explains how design elements were mapped to UML2 elements.

Element	Description
Packages	Packages are mapped to UML2 `Package` elements. The `Name` attribute has to be set. Other attributes are not used. Nested packages are not allowed. `Package` elements has to be children of the `Model` element.
Classes	Classes are mapped to UML2 `Class` elements. Abstract classes have their `Is Abstract` attribute set to true. The `Name` attribute has to be set. Other attributes of the `Class` element (including `Is Leaf`) are not used. If the class extends other class, `Generalization` subelement is used. If the class implements an interface `InterfaceRealization` subelement is used. Classes are tagged with <<FwClass>> stereotype defined in the FW Profile[3].
Interfaces	Interfaces are mapped to UML2 `Interface` elements. The `Name` attribute has to be set. Other attributes are not used. Value of `Is Abstract` attribute is ignored too – interfaces are always considered abstract. If the interface extends other interface, `Generalization` subelement is used. Interfaces must be tagged with <<FwInterface>> stereotype defined in the FW Profile.
Attributes	Class attributes are mapped to UML2 `Property` elements. The following attributes must be set for `Property` element: `Name`, `Type`, `Visibility`, `Lower Bound`, `Upper Bound`, `Is Static`. Other attributes are not used. `Lower Bound` must be set to 1. If the `Property` as an array or a list, the `Upper Bound` must set to number higher then 1 or to `*`. The star character is used if the size of the array is not know at the design time or if the `Property` is a list.

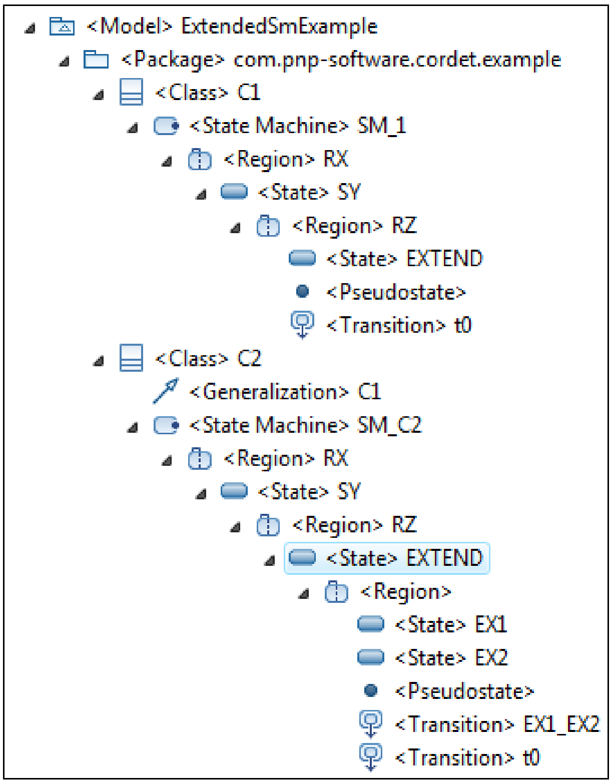
[3] This stereotype should be automatically applied to all classes in the model. Note that the stereotype is also applied to all UML2 elements that extends the Class element, for example StateMachine or OpaqueBehavior.

Element	Description
	It is possible to define a default value using the `Default` attribute. The UML2 Association and UML2 Aggregation elements are not used the model design. Class attributes are used in their place.
Operations	Operations are mapped to UML2 `Operation` elements. The following attributes must be set of `Operation` element: `Name`, `Type`, `Visibility`, `Lower Bound`, `Upper Bound`, `Is Static`, `Is Abstract`, `Is Leaf`. `Lower Bound` must be set to 1. If the `Operation` returns an array or a list, the `Upper Bound` must set to number higher then 1 or to `*`. The star character is used if the size of the array is not know at the design time or if the `Operation` returns a list. Operations may define behavior. If the behavior of the `Operation` is defined in the model, the UML2 `OpaqueBehavior` element must be used. This `OpaqueBehavior` element and `Operation` element must have the same parent (`Class` element). Attribute `Method` of the `Operation` must reference the `OpaqueBehavior` element. Attribute `Specification` of the `OpaqueBehavior` element must reference the `Operation` element. Along with the `Specification` attribute, the `OpaqueBehavior` must also specify the `Body` attribute. The Body attribute contains code in FW Profile Action Language [RD38]. All other attributes of the `OpaqueBehavior` are ignored (including value of the `Language` attribute). Is Abstract attribute of `Operation` elements defined in interfaces is not used. Other attributes of `Operation` are not used. Operations may have parameters which are UML2 `Parameter` elements. The `Parameter` is children of the `Operation`. The following attributes must be set for each Parameter: `Name`, `Type`, `Lower Bound`, `Upper Bound`, `Direction`.
Constructors	Constructors are mapped to UML2 `Operation` elements. For constructors the same restrictions apply as for operations. In addition, the name of the `Operation` must be the same as parent `Class` name. The return type of constructor has to be `void`.
State Machine	State machines are mapped to UML2 `StateMachine` elements. The `StateMachine` element is a subelement of `Class` element. The `Name` attribute has to be set. Other attributes are not used. The `StateMachine` contains one UML2 `Region` element. The `Region` element in turn contains `State`, `Pseudostate` and `Transition` elements. The notion of region is not defined in the Cordet Methodology; it is an UML2 construct only. From this perspective it can be seen as a container holding set of `State`, `Pseudostate` and `Transition` elements, while (by definition) each state, pseudo state and transition must have `Region` as its parent. Region thus is merely a mediator between state machine and its states, pseudo states and transitions. State machines must be tagged with <<FwStateMachine>> stereotype defined in the FW Profile.
State	States are mapped to UML2 `State` elements.

Element	Description
	The State elements are subelements of Region elements. The Name attribute has to be set. Other attributes are not used. State may have an *entry action* and/or an *exit action*. Both actions are OpaqueBehavior subelements of the State element. The Body attribute of these respective OpaqueBehavior elements must contain code in FW Profile Action Language. States are tagged with <<FwState>> stereotype defined in the FW Profile.
Transitions	Transitions are mapped to UML2 Transition elements. The Transition elements are subelements of Region elements. No attributes are used, but it is recommended to name the transition using the Name attribute (following the naming conventions). Transitions element must have one trigger subelement. The only exception are transitions that originate in initial, history, or choice states. Transitions may have guard and transition action. Transitions are tagged with <<FwTransition>> stereotype defined in the FW Profile.
Guard	Guards are mapped to UML2 Constraint elements. The Constraint element must be subelement of the Transition element. The Constraint element must contain an OpaqueBehavior element. The Body attribute of the OpaqueBehavior element must contain code in FW Profile Action Language. Other attributes of Constraint or OpaqueBehavior are not used.
Transition actions	Transition actions are mapped to UML2 OpaqueBehavior elements. The OpaqueBehavior element must be subelement of the Transition element. The Body attribute of the OpaqueBehavior element must contain code in FW Profile Action Language. Other attributes are not used.
Triggers	Each trigger is mapped to three UML2 elements at once: Trigger, CallEvent, and Operation. For each trigger in the state machine an Operation element of the same name has to be defined in the Class where the state machine is defined. This operation must have <<FwTrigger>> stereotype applied, has to be final (Is Leaf attribute must be set to true), its return type must be void, and the operation cannot have attached OpaqueBehavior. For each trigger in the state machine an CallEvent element has to be defined in the Package where the Class containing the state machine is defined. The Operation attribute of the CallEvent element has to reference the trigger operation. The Name attribute of the CallEvent element has to be set. For each trigger in the state machine a Trigger has to be added as subelement of the transition. The Event attribute of the Trigger element has to reference the CallEvent element above. The Name attribute of the CallEvent element has to be the same as the name of the trigger Operation above.
Embedded state	Embedded state machines are mapped to UML2 StateMachine

Element	Description
machines	elements with the following structure.

Element	Description
	Given state machine SM_C1 associated to class C1 and state machine SM_C2 associated to class C2 and given that C2 extends C1 and SM_C2 extends state S defined in SM_C1. The mapping to UML2 is as follows. Between state machine SM_C1 element and `State` S1 element there is a unique sequence of `Region` elements and `State` elements. In order to implement the extension in class C2, the state machine SM_C2 has to be created. Then the sequence of `Region` elements and `State` elements has to be created in SM_C2 as an exact copy of the sequence as it exists in SM_C1 (the same names of regions and states). This process ends with `State` S2 defined in SM_C2 the has the same name as S1. At this point new `Region` element is added as subelement of `State` S2. Further, new `State`, `PseudoState`, and `Transition` elements are added as subelements of the region. This way the S2 state was extended. The figure 4.4-2 illustrates this process. In the figure the name of both S1 and S2 states is EXTEND.
Comments	Description of classes, interfaces, operations, and attributes can be added to the UML2 model using `Comment` elements. The `Comment` element is always a child of the element being commented. The `Body` attribute holds the comment itself which is a plain text that may contain HTML tags. Is is also possible to use several special Javadoc-like tags to describe e.g. operation parameters or return values. There is a documentation generator that extracts the comments from the model and build design documentation in a form of a website. The tool is documented in [RD38]. UML2 allows comments to be added to any UML2 element, however, the tool only process comments attached to classes, interfaces, operations, and attributes.



```
<Model> ExtendedSmExample
  <Package> com.pnp-software.cordet.example
    <Class> C1
      <State Machine> SM_1
        <Region> RX
          <State> SY
            <Region> RZ
              <State> EXTEND
              <Pseudostate>
              <Transition> t0
    <Class> C2
      <Generalization> C1
      <State Machine> SM_C2
        <Region> RX
          <State> SY
            <Region> RZ
              <State> EXTEND
                <Region>
                  <State> EX1
                  <State> EX2
                  <Pseudostate>
                  <Transition> EX1_EX2
                  <Transition> t0
```

Fig. 4.4-2: Mapping of extended state machines

5 DH FRAMEWORK ARCHITECTURE

This section describes the architecture of the DH Framework. The framework architecture is described in terms of the main components that are defined by the framework.

This section is intended to place the reader in a better position to understand the documentation attached to the UML2 design model.

5.1 Target Domain

The target domain of the DH Framework is defined in RD37. For the convenience of the reader, an outline of this domain is presented below.

The core of a DH subsystem is the handling of incoming telecommands and the generation of outgoing telemetry data. A framework approach for the DH subsystem is only possible if the format and content of the telecommands and telemetry data is, at least to some extent, standardized.

The DH Framework assumes that telecommanding and telemetry is implemented in accordance with the Packet Utilization Standard or PUS [RD-36]. Applicability to non-PUS systems is not excluded (see discussion in RD37) but the PUS constitutes the conceptual framework within which the DH Framework is defined.

The PUS defines the external interface of a DH application in terms of the *services* that the application must provide to other applications. The services are in turn defined in terms of *telecommand packets* that the application must be able to handle, and *telemetry packets* that the application must be able to generate.

The PUS implicitly defines the concept of an abstract telecommand packet and an abstract telemetry packet. This concept is independent of the particular service which the telecommand packet or telemetry packet supports. The definition of the abstract telecommand packets and abstract telemetry packets covers the features that are common to all PUS-compliant telecommand packets and PUS-compliant telemetry packets.

The DH Framework provides software interfaces and software components that support the implementation and manipulation at software level of abstract telecommand packets and abstract telemetry packets.

The DH Framework, in other words, transposes the PUS to the software level. The PUS standardizes the services to be provided by a DH application. The DH Framework standardizes the software interfaces through which those services are accessed at software level within a DH application.

The PUS also defines a taxonomy of specific services that may be provided by a DH application. Each kind of service is identified by a *type.* The provision of that service by the on-board application is supported by a number of telecommand and telemetry packets. Each kind of telecommand packet or telemetry packet within the service type is identified by a *subtype*.

The PUS pre-defines some service types. These pre-defined service types represent services that are commonly used in on-board systems. For these pre-defined services, the PUS defines both the physical layout and the functional interpretation of the associated telecommand and telemetry packet subtypes.

Application designers can use the services pre-defined by the PUS or they can defined new application-specific services.

The DH Framework does not support the implementation of the pre-defined PUS service types. Provision of this kind of support would be technically feasible and industrially desirable but it is outside the scope of the CORDET Project.

Future extensions of the DH Framework might extend the interfaces and components provided by the framework to support some or all of the PUS service types.

The DH Framework also does not support the routing of telecommand and telemetry packets between applications. The framework is aimed at the *processing* of telecommands *within* a PUS application and at the *generation* of telemetry packets from *within* a PUS application. It does not explicitly cover the transmission of packets *between* PUS applications.

The concepts provided by the framework, however, could be used to implement such a dispatching framework to link together PUS applications. In particular, the support offered by the framework for the manipulation of abstract telecommands and telemetry packets would facilitate the implementation of an application-independent infrastructure for routing telecommand and telemetry packets.

5.2 Design Heritage

The design of the DH Framework is based on the design of the so-called ETH Demonstrator Framework[4] developed at ETH by the authors of this technical note in the ASSERT Project (see [RD29] and [RD30]).

The design of the DH Framework differs from the design of the ETH Demonstrator Framework in the following respects:

- The DH Framework conforms to the naming conventions defined in section 4.1. This has led to several changes in names of UML2 elements.
- The ETH Demonstrator Framework conformed to the FW Profile as it had been defined in the ASSERT Project. In the CORDET Project, this profile was extended and the DH Framework conforms to this extended profile[5].
- The DH Framework eliminates the `IManagedMemory` interface and the associated `ManagedMemory` component. In the ETH Demonstrator Framework, this component was one of the “basic components” of the framework.
- In order to compensate the elimination of the `IManagedMemory` interface and the associated `ManagedMemory` component, the state machine of the telecommand component was modified with the introduction of a new state.
- Also in order to compensate the elimination of the `IManagedMemory` interface and the associated `ManagedMemory` component, the state machine of the telecommand component was modified with the introduction of two new triggers (`putInUse` and `putOutOfUse`).
- The trigger operations on class `Telecommand` has been simplified. This also had an impact on the implementation of the telecommand loader.
- The `TelecommandStream` interface has been substantially re-designed with the addition of several new methods

With reference to the elimination of the `IManagedMemory` interface and the associated `ManagedMemory` component, it should be stressed that the function of the old

[4]

[5] The extended FW Profile is available from: http://www.pnp-software.com/fwprofile. This web site also describes the extension to the original FW Profile.

MemoryManaged component is covered in the new FW by the new state NOTUSED. In other words, the state machine hierarchy has been "flattened" with the elimination of one layer and the introduction of one extra state. There is no therefore loss of functionality: the MemoryManaged component was used to manage the usage state of a component by a factory and the NOTUSED state has exactly the same function

5.3 Design Pattern Heritage

The design patterns used for the definition of the DH Framework design are the same as were defined for the telecommand and telemetry management functions of the OBS Framework [RD18]. Some of these design patterns are based on the design patterns of RD7. This in particular applies to the Telecommand Pattern that is based on the Command Design Pattern of RD7.

5.4 High-Level Functions

The components and interfaces defined by the DH Framework can be allocated to the following high-level functions:

- Telecommanding Function: this function is responsible for the management of incoming telecommands.
- Telemetry Function: this function is responsible for the management of outgoing telemetry packets.
- Initialization and Configuration Function: this function is responsible for the management of the initialization and configuration process of other framework components (this purpose of this function is clarified in section 5.7).

Each framework component or framework interface is allocated to one and only one function. The UML2 model of the DH Framework defines a dedicated package for each function as indicated in table 5.4.1.

Each function is described in greater detail in a dedicated section below.

Table 5.4-1: Allocation of Functions to Packages

Function	Package
Telecommanding Function	Telecommand Package
Telemetry Function	Telemetry Package
Initialization and Configuration Function	Basic Package

5.5 Telecommanding Function

Figure 5.5-1 shows the architecture of the telecommanding function in an informal notation. The boxes represent components or interfaces provided by the framework. The arrows represent flows of control or of data.

The TcStream component is the primary interface between the telecommanding function and the application within which it is embedded. It acts as a data stream from which raw telecommand packets can be acquired. Conceptually, it resembles a Java input stream.

The TcStream is characterized in the DH Framework by interface ITelecommandStream. The framework also defines a class implementing this interface and implementing the invariant behaviour of TcStream (class TelecommandStream). Applications should extend this class to implement their own TcStream. An application can only have one single instance of TcStream.

Telecommands are manipulated within an application instantiated from the DH Framework as components implementing interface `ITelecommand`. The de-serialization of a raw telecommand (the creation of an instance of interface `ITelecommand` implementing the telecommand defined in a sequence of bytes received from the TcStream) is done by the TcLoader component.

The TcLoader is characterized in the DH Framework by interface `ITelecommandLoader`. The framework also defines a class implementing this interface and implementing the invariant behaviour of TcLoaders (class `TelecommandLoader`). Applications should extend this class to implement their own TcLoader. An application can only have one single instance of TcLoader.

The TcLoader loads the telecommand components into TcManager components which are then responsible for controlling the execution of the telecommand. An application may have one or more TcManager components (for instance, it may have one TcManager for each telecommand priority level).

TcManager components are characterized by interface `ITelecommandManager` for which the framework provides an implementation as class `TelecommandManager`. The TcManager components are provided by the framework as final components that would not normally need to be extended by applications (although applications can always decide to create their own implementation of interface `ITelecommandManager`).

A TcFactory dynamically provides unconfigured telecommand components to the TcLoader. After telecommands have completed their execution (or after they have been aborted), the TcManager returns the components encapsulating them to the TcFactory.

TcFactory components are characterized by interface `ITelecommandFactory`. No invariant behaviour can be associated to this interface and therefore no default or partial implementation for this interface is provided by the framework.

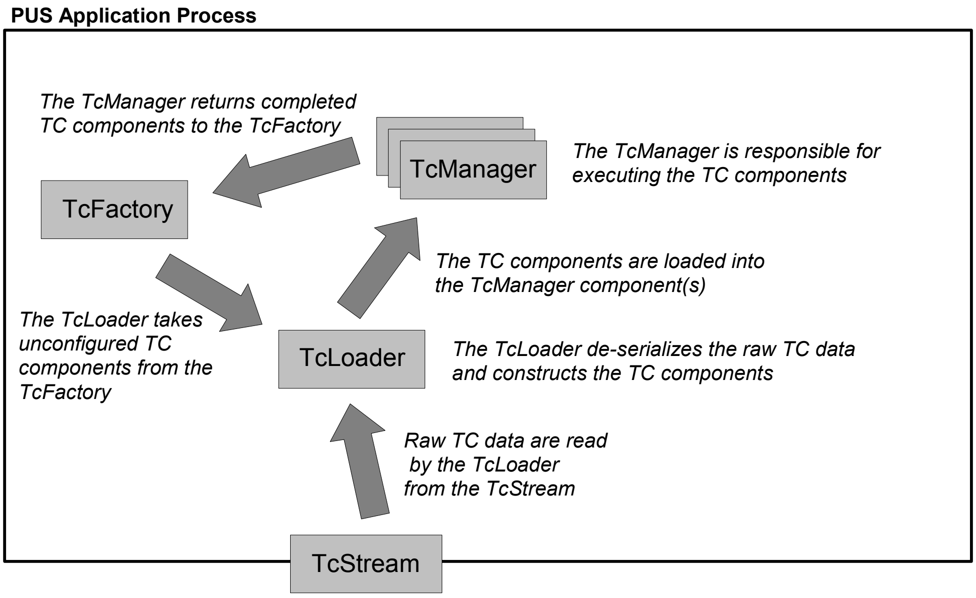

PUS Application Process
The TcManager returns completed TC components to the TcFactory
TcManager
The TcManager is responsible for executing the TC components
TcFactory
The TC components are loaded into the TcManager component(s)
The TcLoader takes unconfigured TC components from the TcFactory
TcLoader
The TcLoader de-serializes the raw TC data and constructs the TC components
Raw TC data are read by the TcLoader from the TcStream
TcStream

Fig. 5.5-1: Architecture of Telecommanding Function

5.6 Telemetry Function

Figure 5.6-1 shows the architecture of the telemetry function in an informal notation. The boxes represent components or interfaces provided by the framework. The arrows represent flows of control or of data.

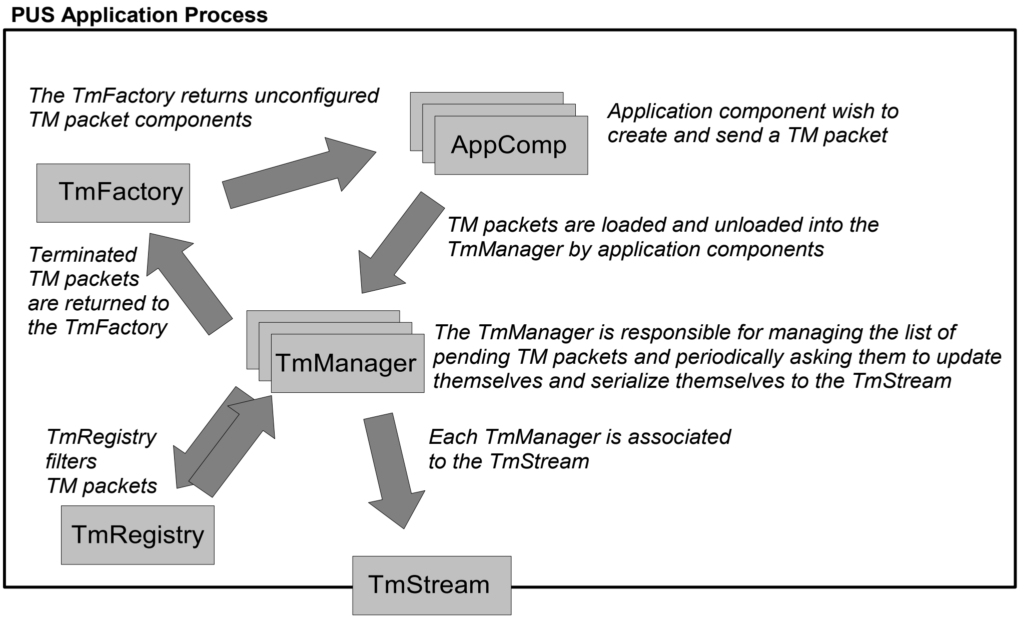

PUS Application Process
The TmFactory returns unconfigured TM packet components
AppComp
Application component wish to create and send a TM packet
TmFactory
TM packets are loaded and unloaded into the TmManager by application components
Terminated TM packets are returned to the TmFactory
TmManager
The TmManager is responsible for managing the list of pending TM packets and periodically asking them to update themselves and serialize themselves to the TmStream
TmRegistry filters TM packets
Each TmManager is associated to the TmStream
TmRegistry
TmStream

Fig. 5.6-1: Architecture of Telemetry Function

The TmStream component is the primary interface between the telemetry function and the application within which it is embedded. It acts as a data stream to which telemetry packets are serialized. Conceptually, it resembles a Java output stream.

The TmStream is characterized in the DH Framework by interface `ITelemetryStream`. The framework also defines a class implementing this interface and implementing the invariant behaviour of TmStream (class `TelemetryStream`). Applications should extend this class to implement their own TmStream. An application can only have one single instance of TmStream.

Telemetry packets are manipulated within an application as components implementing interface `ITelemetry`. When an application component finds that it needs to generate a telemetry packet, it asks for an unconfigured instance of a telemetry packet component of the right type and subtype from the TmFactory. It then configures it and hands it over to a TmManager component.

Telemetry packet components are capable of serializing themselves to a TmStream. The TmManager is responsible for managing the serialization process. It buffers up the telemetry packets it receives from the application components and directs them to start the serialization process.

TmManager components are characterized by interface `ITelemetryManager` for which the framework provides an implementation as class `TelemetryManager`. The TmManager components are provided by the framework as final components that would not normally need to be extended by applications (although applications can always decide to create their own implementation of interface `ITelemetryManager`).

In an application there may be several TmManager components corresponding to different levels of priorities for handling the telemetry packets.

A telemetry packet can only be loaded onto one TmManager. If the same telemetry packets must be sent to two destinations (e.g. to ground and to an on-board log), then either two identical packets with the same content must be created or a TmStream component must be provided that encapsulates two telemetry destinations.

Some telemetry packets are sporadic (they only need to be sent once – e.g. event reports) whereas others are cyclical (they must be sent more than once – e.g. housekeeping packets). The TmManager returns sporadic packets to the TmFactory immediately after they have been serialized to the TmStream and it keeps cyclical packets in its queue of pending packets for repeated serialization. Cyclical telemetry packets are capable to updating their own content in between successive serializations. The TmManager is responsible for controlling the update process and for correctly interleaving it with the serialization process.

Telemetry packets may be subject to a filtering process that changes their internal state while they are being processed by the TmManager. Filtering information is stored in the TmRegistry. The TmManager passes telemetry packet to the TmRegistry before processing it. This gives the TmRegistry the opportunity to change the internal state of the telemetry packet according to the information it contains.

Thus, for instance, if a telecommand wishes to disable a housekeeping telemetry packet, then the telecommand would provide the information about the packet to the TmRegistry (it would for instance provide the packet type and subtype and its SID) and then the TmRegistry would reconfigure the telemetry packet when it receives it from the TmManager. In this way, telemetry packets and the components that change their internal state (typically telecommands) are decoupled.

The TmRegistry component is characterized by interface `ITelemetryRegistry`. No default or partial implementation of this interface is provided by the DH Framework.

A TmFactory dynamically provides unconfigured telemetry packet components to the application components that need to send telemetry data to the ground. After telemetry packets have been serialized to the TmStream, the TmManager returns the components encapsulating them to the TmFactory where they remain available for use by other application components.

The TmFactory component is characterized by interface `ITelemetryFactory`. No default or partial implementation of this interface is provided by the DH Framework.

Note finally that, as already mentioned above and for reasons discussed at greater length in RD37, an instantiation of the DH Framework can only have one single TmStream component. Hence, routing to the destination is done by the application-specific TmStream instance (namely it is done by creating an instance of interface TmStream that "knows" about how to route packets to their destination).

5.7 Initialization and Configuration Function

The DH Framework assumes that framework components, when they are instantiated at application initialization time, are completely unconfigured. In other words, it is assumed that framework components do not contain any default values for their internal state variables. This is a reasonable assumption in view of the fact that the components are intended to be deployed in different contexts and in different applications.

The initialization and configuration function supports the process whereby a framework component is made ready for operational use by the application instantiated from the framework.

Since the initialization and configuration process must in principle be performed by all framework components, the framework maps it to an interface – interface `IComponent` – that must be implemented by all other framework interfaces.

The framework provides an implementation of interface `IComponent` – class `Component` – that implements the initialization and configuration process.

The DH Framework makes a distinction between *initialization* and *configuration*. Components are first initialized and then configured. Initialization is an irreversible process where the values of parameters that can only be set once is defined. Typically, in the initialization process, the memory for the component's internal data structure is allocated.

Configuration is a process where the values of parameters that can be dynamically changed is first defined. After a component is configured, it is ready to start its normal operation.

Component configuration must therefore be performed by the application initialization code by loading values for all the component parameters. The component may perform legality checks on the loaded values and, once all values have been loaded and their legality has been confirmed, then the component is declared to be configured.

Components can be re-configured at run-time (but they cannot be re-initialized).

6 DH FRAMEWORK – DOMAIN DICTIONARY MAPPING

This section describes how the domain dictionary entries for the DH Framework are mapped to elements in the UML2 design model of the framework.

The mapping is described by means of tables that, on the left-hand side, list the dictionary entry together with its *attributes*, *operations*, *actions*, and *checks* and, on the right-hand side, list the UML2 elements to which they are mapped. One table is provided for each dictionary entry.

In most cases, the following general mapping rules apply:

- Dictionary entries are mapped to interfaces.
- The attributes of a dictionary entry are mapped either to getter methods defined on the interface associated to the dictionary entry, or to UML2 attributes in the class that implements the interface in the framework.
- The operations of a dictionary entry are mapped to non-virtual methods that are defined on the interface associated to the dictionary entry,
- The actions of a dictionary entry are mapped to virtual methods that are defined in the class that implements the interface in the framework.
- The checks of a dictionary entry are mapped to virtual methods that are defined in the class that implements the interface in the framework.

In the case of operations, actions, and checks, the mapping is normally to one single method in the UML2 model for each operation/action/check. In some cases, however, one operation/action/check is mapped to a set of related methods defined on the same class or interface in the UML2 model.

The next two sections define the mapping for the dictionary entries defined for the telcommanding and telemetry concepts.

6.1 Mapping of Dictionary Entries for Telecommand Concept

This section defines the mapping to design elements for the domain dictionary entries related to the framework telecommand concept.

Dictionary Term	**Framework Telecommand**	**Interface `ITelecommand`**
Attributes	Type	`pusType` attribute in `Telecommand`
	Subtype	`pusSubType` attribute in `Telecommand`
	Source sequence counter	`sourceSequenceCounter` attribute in `Telecommand`
	Telecommand State	is* methods in ITelecommand
Operations	Execute	`ready`, `accept`, and `execute` methods in `ITelecommand`
	Abort	`abort` method in `ITelecommand`
Actions	Start Action	`doStart` method in `Telecommand`
	Progress Action	`doProgress` method in `Telecommand`
	Completion Action	`doComplete` and `doFinalize` methods in `Telecommand`

	Abort Action	`doAbort` method in `Telecommand`
Checks	Acceptance Check	`doAcceptanceCheck` method in `Telecommand`
	Ready Check	`doReadyCheck` method in `Telecommand`
	Start Check	`canStart` method in `Telecommand`
	Progress Check	`canProgress` and `hasProgressFailed` methods in `Telecommand`
	Completion Check	`canProgress` and `hasProgressFailed` methods in `Telecommand`

Dictionary Term	**Telecommand Manager**	**Interface `ITelecommandManager`**
Attributes	List of Pending Telecommands	`tcList` attribute in `TelecommandManager`
Operations	Activate	`activate` method in `TelecommandManager`
	Abort a Pending Telecommand	`abort` method in `TelecommandManager`
	Abort all Pending Telecommands	Not mapped.
	Load a new Telecommand	`load` method in `TelecommandManager`

Dictionary Term	**Telecommand Loader**	**Interface `ITelecommandLoader`**
Attributes	List of Telecommand Managers	`TcManagerList` attribute in `TelecommandLoader`
	Telecommand Stream	`tcStream` attribute in `TelecommandLoader`.
Operations	Activate	`activate` method in `TelecommandLoader`
Actions	Telecommand Loading Action	`loadIntoTcManager` method in `TelecommandLoader`

Dictionary Term	**Telecommand Stream**	**Interface `ITelecommandStream`**
Operations	Query for the presence of a new packet	`isPacketArrived` method in `ITelecommandLoader`
	Read raw telecommand data	`read*` methods in `ITelecommandLoader`

6.2 Mapping of Dictionary Entries for Telemetry Concept

This section defines the mapping to design elements for the domain dictionary entries related to the framework telemetry concept.

Dictionary Term	**Framework Telemetry Packet**	**Interface `ITelemetry`**
Attributes	Type	`pusType` attribute in `Telemetry`
	Subtype	`pusSubType` attribute in `Telemetry`
	Destination	`destination` attribute in `Telemetry`
	Time Stamp	`timeStamp` attribute in `Telemetry`
Operations	Execute	`Update`, `serialize` and `terminate` methods in `Telemetry`
	Configure	`configure` method in `Telemetry`
	Enable/Disable	`setEnable` method in `Telemetry`
Actions	Configuration Action	`doConfigure` method in `Telemetry`
	Serialization Action	`doSerialize` method in `Telemetry`
	Update Action	`doUpdate` method in `Telemetry`
	Termination Action	`doTerminate` method in `Telemetry`
Checks	Enable Check	`isEnabled` method in `Telemetry`
	Termination Check	`canTerminate` method in `Telemetry`
	Hold Check	`isHeld` method in `Telemetry`

Dictionary Term	**Telemetry Manager**	**Interface `ITelemetryManager`**
Attributes	List of Pending Telemetry Packets	`tmList` attribute in `TelemetryManager`
Operations	Activate	`activate` method in `TelemetryManager`
	Load a new Telemetry Packet	`load` method in `TelemetryManager`

Dictionary Term	**Telemetry Stream**	**Interface `ITelemetryStream`**
Operations	Query for readiness to receive data	`isReady` method in `TelemetryStream`
	Write raw telemetry data	`writeParam` method in `TelemetryStream`

7 DH FRAMEWORK – MAPPING OF SHARED PROPERTIES

This section describes how the shared properties defined at domain analysis level for the DH Framework have been mapped to the UML2 model of the framework.

Each shared property stated in the domain model is translated into a functional property on the UML2 model of the DH Framework. These functional properties could in principle be formally verified on the framework model.

For the convenience of the reader, the domain model properties are re-stated here in the same form in which they were stated in [RD-37] and the formulation of the design-level functional property is given immediately afterwards. The format is therefore as follows:

<Property Identifier>	<Statement of Domain Model Property>
	<Statement of Domain Design Property>

Some properties depend on certain pre-conditions being satisfied. In such cases, the pre-condition is stated as part of the design-level properties.

7.1 Shared Properties for Telecommand Concept

This section describes the mapping of shared properties associated to the telecommand-related entries in the domain dictionary of the DH Framework.

7.1.1 Telecommand Execution

The properties relative to the telecommand execution are implemented in the logic of the state machine associated to the `Telecommand` class. Some of these properties additionally depend on the state machine being triggered in certain ways. This pre-condition is captured by assuming that the telecommand component is controlled by a telecommand manager.

In order to understand the mapping from the domain model to the design model properties, the following points should be born in mind:

- The domain model `execute` operation is mapped in the design model to the methods `accept`, `ready`, and `execute`.
- The domain model state COMPLETED is mapped in the design model to the state machine states COMPLITING and TERMINATED

P7.1.1-1	*A telecommand can change its internal state only in response to an execute or to an abort operation being performed upon it.*
	The state machine associated to class `Telecommand` has the following trigger methods: `abort`, `accept`, `ready`, `execute`.
P7.1.1-2	*When a telecommand is executed for the first time, it enters state ACCEPTED if its acceptance check is passed, otherwise it is aborted.*
	Precondition: the telecommand component is in state RECEIVED and has been loaded into a telecommand manager. *If method `activate` is called on the telecommand manager, then the telecommand state changes to ACCEPTED if method `isAcceptanceCheck` return TRUE, otherwise it changes to ABORTED.*
P7.1.1-3	*If an ACCEPTED telecommand is executed, it performs its ready check and, if this is passed, it attempts to enter state STARTED. If it is not successful, it*

	remains in state ACCEPTED.
	Precondition: the telecommand component is in state ACCEPTED and has been loaded into a telecommand manager. *If method `activate` is called on the telecommand manager, then the telecommand state changes to STARTED if method `isReadyCheckOK` returns TRUE, otherwise it re-enters state ACCEPTED.*
P7.1.1-4	*A telecommand can enter state INPROGRESS only if its start check is passed.*
	A necessary condition for state INPROGRESS in the state machine of class `Telecommand` to be entered is that method `canStart` returns TRUE when the transition into INPROGRESS is attempted.
P7.1.1-5	*If a STARTED telecommand is executed, it performs the progress check and, if this is passed, the telecommand enters state INPROGRESS, otherwise it is aborted.*
	Precondition: the telecommand component is in state STARTED and has been loaded into a telecommand manager. *If method `activate` is called on the telecommand manager, then the telecommand state changes to ABORTED if method `canProgress` returns FALSE.*
P7.1.1-6	*If an INPROGRESS telecommand is executed, it performs the progress check and, depending on its outcome, it either re-enters state INPROGRESS, or it attempts to enter state COMPLETED, or else it is aborted.*
	Precondition: the telecommand component is in state INPROGRESS and has been loaded into a telecommand manager. *If method `activate` is called on the telecommand manager, then the telecommand state changes to ABORTED if its method `hasProgressFailed` returns TRUE, else it stays in state INPROGRESS if its method `canProgress` returns TRUE, or it changes to COMPLETED if method `canProgress` returns FALSE.*
P7.1.1-7	*A telecommand can enter state COMPLETED only if its completion check is passed. Otherwise it is aborted.*
	A necessary condition for state TERMINATED in the state machine of class `Telecommand` to be entered is that method `canComplete` returns TRUE when the transition into state TERMINATED is attempted.
P7.1.1-8	*When a telecommand becomes STARTED, it executes its start action.*
	When state STARTED in the state machine of class `Telecommand` is entered, then method `doStart` is executed.
P7.1.1-9	*Every time a telecommand enters state INPROGRESS, it executes its progress action.*
	When state INPROGRESS in the state machine of class `Telecommand` is entered, then method `doProgress` is executed.
P7.1.1-10	*When a telecommand becomes COMPLETED, it executes its completion action.*
	When state COMPLETED in the state machine of class `Telecommand` is entered, then method `doCompleted` is executed.
P7.1.1-11	*When a telecommand is aborted, it executes its abort action.*
	When state ABORTED in the state machine of class `Telecommand` is entered, then method `doAbort` is executed.
P7.1.1-12	*Execution of a telecommand that is aborted or COMPLETED has no effect.*
	The trigger methods `accept`, `ready`, and `execute` have no effect when the

	state machine of class `Telecommand` is in state ABORTED or TERMINATED.

7.1.2 Telecommand Management

The properties relative to the telecommand management are implemented in the methods of class `TelecommandManager`. The implementation of these methods is modelled using the action language of the FW Profile.

P7.1.2-1	If a telecommand is loaded in a telecommand manager, then the telecommand is added to the list of pending telecommands.
	Precondition: the array `tcList` has at least one NULL element. If method `load` is called on `TelecommandManager`, then its argument is added to `tcList`.
P7.1.2-2	When a telecommand manager is activated, it executes all the telecommands in its list of pending telecommands.
	If method `activate` is called on `TelecommandManager`, then the methods `accept`, `ready`, `execute` are called in sequence on each non-null entry in `tcList`.
P7.1.2-3	When a telecommand manager is activated, it removes from the list of pending telecommands the telecommands that have been aborted or that are in state COMPLETED.
	If method `activate` is called on `TelecommandManager`, then telecommand entries in `tcList` that are in state ABORTED or TERMINATED are removed from the list.
P7.1.2-4	When a telecommand manager is asked to abort one of its pending telecommands, it aborts the telecommands and removes it from its list of pending telecommands.
	Precondition: the argument of method `abort` is in `tcList`. If method `abort` is called on `TelecommandManager`, then method `abort`, is called on its argument and the argument is removed from `tcList`.

7.1.3 Telecommand Loading

The telecommand loader reads the raw telecommand data from the telecommand stream, uses them to build the corresponding framework telecommand component, and loads the newly created telecommand component onto a telecommand manager.

Since there can be several telecommand managers within the same application, the telecommand loader implements the logic that decides where each telecommand should be loaded. This logic is encapsulated in the telecommand loading action.

The telecommand loader reads the raw telecommand data when it is activated. This should not be taken to imply that a polling mechanism must be used to collect telecommands since the activation signal might be linked to the arrival of a new raw telecommand. The logic that decides when to activate the telecommand loader is outside the DH Framework. In this sense, the DH Framework neither enforces nor assumes a particular mechanism for detecting and responding to the arrival of raw telecommands.

P7.1.3-1	When a telecommand loader is activated, it reads the raw telecommand data (if any are available) from the telecommand stream, it creates the framework telecommand component, and it executes the telecommand loading action.
	Precondition: the telecommand stream is in state PACKET_ARRIVED when

	method `activate` is called on the telecommand loader. *When method `activate` is called on `TelecommandLoader`, then a telecommand instance is retrieved from `TelecommandFactory`, its method `deserialize` is called, and then method `loadIntoTcManager` is called.*

7.2 Shared Properties for Telemetry Concept

This section describes the mapping of shared properties associated to the telemetry-related entries in the domain dictionary of the DH Framework.

7.2.1 Telemetry Packet Configuration

The properties relative to the telemetry packet configuration are implemented in the logic of the state machine associated to the `Telemetry` class.

P7.2.1-1	*When a telemetry packet is configured, it performs its configuration action.*
	When state PACKET_CONFIGURE in the state machine associated to class `Telemetry` is entered, then method `doConfigured` is executed.

7.2.2 Telemetry Packets Execution

The properties relative to the telemetry packet execution are implemented in the logic of the state machine associated to the `Telecommand` class. Some of these properties additionally depend on the state machine being triggered in certain ways. This pre-condition is captured by assuming that the telecommand component is controlled by a telemetry manager.

In order to understand the mapping from the domain model to the design model properties, the following points should be born in mind:

- The domain model `execute` operation is mapped in the design model to the methods `update`, `serialize`, and `terminate`.

P7.2.2-1	*Only telemetry packets that have been configured can be executed.*
	States PACKET_UPDATE and PACKET_SERIALIZE in the state machine associated to class `Telemetry` can only be entered after state PACKET_CONFIGURED has been entered.
P7.2.2-2	*When a telemetry packet is executed, it performs its enable check to verify whether the packet is enabled.*
	State PACKET_UPDATE in the state machine associated to class `Telemetry` can only be entered if method `isEnabled` returns TRUE when the transition into the state is attempted.
P7.2.2-3	*When a telemetry packet is executed, it performs its hold check to verify whether the packet is being held.*
	Precondition: the telemetry packet component is in state PACKET_SERIALIZE or in state PACKET_HOLDING and it has been loaded into a telemetry manager. *If method `activate` is called on the telemetry manager, then state PACKET_UPDATE can only be entered if method `isHeld` returns FALSE.*
P7.2.2-4	*Execution of a telemetry packet that is neither disabled nor held results in the telemetry packet performing first its update action, and then its serialization action.*
	Precondition: the telemetry packet component is in state PACKET_SERIALIZE or in state PACKET_HOLDING and it has been loaded into a telemetry

	manager. *If method `activate` is called on the telemetry manager, and if its methods isEnabled and isHeld return, respectively, TRUE and FALSE, then the telemetry packet goes through states PACKET_UPDATE and PACKET_SERIALIZE.*
P7.2.2-5	*When a telemetry packet is serialized, it writes its content to the telemetry stream associated to the telemetry manager that executes the packet.*
	When state PACKET_SERIALIZE in the state machine associated to class `Telemetry` is entered, then its method `doSerialize` is executed.
P7.2.2-6	*After performing the serialization action, a telemetry packet performs its termination check to verify whether it is terminated.*
	Precondition: the telemetry packet component is in state PACKET_SERIALIZE or in state PACKET_HOLDING and it has been loaded into a telemetry manager. *If method `activate` is called on the telemetry manager, and if its method canComplete returns TRUE then the telemetry packet enters state TERMINATED.*
P7.2.2-7	*If the termination check indicates that the packet is terminated, then the telemetry packet executes its termination action.*
	When state TERMINATED in the state machine associated to class `Telemetry` is entered, then its method `doTerminate` is executed.

7.2.3 Telemetry Packet Management

The properties relative to the telemetry packet management are implemented in the methods of class `TelemetryManager`. The implementation of these methods is modelled using the action language of the FW Profile.

P7.2.3-1	*If a telemetry packet is loaded in a telemetry manager and if the packet is configured, then the telemetry packet is added to the list of pending telemetry packets.*
	Precondition: the array `tmList` has at least one NULL element. *If method `load` is called on `TelemetryManager`, and its argument is in state PACKET_CONFIGURED, then its argument is added to `tmList`.*
P7.2.3-2	*When a telemetry packet is activated, it executes all the telemetry packets in its list of pending telemetry packets.*
	If method `activate` is called on `TelemetryManager`, then the methods `update`, `serialize`, `terminate` are called in sequence on each non-null entry in `tmList`.
P7.2.3-3	*When a telemetry manager is activated, it removes from the list of pending telemetry packets the packets that, after their execution, are terminated.*
	If method `activate` is called on `TelemetryManager`, then telecommand entries in `tmList` that are in state TERMINATED are removed from the list.

8 DH FRAMEWORK – MAPPING OF FACTORS OF VARIATION

This section describes how the factors of variation defined at domain analysis level for the DH Framework are mapped to the UML2 design model of the framework.

Generally speaking, DH Framework factors of variation are mapped either to virtual methods in one of the framework classes or to framework interfaces for which no implementation is provided by the framework. In some cases, one single factor of variation may be mapped to more than one methods in the same class. Where no default implementation is foreseen for the factor of variation, the mapping is either to a pure virtual method or to an interface.

For the convenience of the reader, the domain model factors of variation are re-stated here in the same form in which they were stated in [RD-37] and the mapping to the UML2 element is added immediately afterwards. The format is therefore as follows:

<Identifier>	<Name of the Factor of Variation>
Description	<Description of the Factor of Variation>
Default	<Default Value of the Factor of Variation>
Range	<Legal Range of the Factor of Variation>
Mapping	<Mapping to UML2 Element in Design Model>

8.1 Attributes as Factors of Variation

All the attributes attached to framework classes represent implicit factors of variation since attribute values are intended to be defined by the application designer at framework instantiation time.

Factors of variations linked to attributes are regarded as trivial and implicitly defined by the domain model and are therefore not further considered in this section.

8.2 Factors of Variation for Telecommand Concept

This section describes the mapping for the factors of variation associated to the telecommand concept.

FV8.2-1	**Telecommand Start Action**
Description	The implementation of the start action used in property P7.1.1-8 is application-specific.
Default	The default implementation of the start action returns without taking action.
Range	Unrestricted.
Mapping	Method `doStart` in class `Telecommand`

FV8.2-2	**Telecommand Progress Action**
Description	The implementation of the progress action used in property P7.1.1-9 is application-specific.
Default	There is no default implementation for the progress action.
Range	Unrestricted.

Mapping	Method `doProgress` in class `Telecommand`

FV8.2-3	**Telecommand Completion Action**
Description	The implementation of the completion action used in property P7.1.1-10 is application-specific.
Default	The default implementation of the completion action returns without taking action.
Range	Unrestricted.
Mapping	Method `doComplete` in class `Telecommand`

FV8.2-4	**Telecommand Abort Action**
Description	The implementation of the abort action used in property P7.1.1-11 is application-specific.
Default	The default implementation of the abort action returns without taking action.
Range	Unrestricted.
Mapping	Method `doAbort` in class `Telecommand`

FV8.2-4	**Telecommand Acceptance Check**
Description	The implementation of the acceptance check used in property P7.1.1-2 is application-specific.
Default	The default implementation of the acceptance check returns TRUE (telecommand is successfully accepted).
Range	This check must return either TRUE (telecommand is successfully accepted) or FALSE (telecommand is not accepted).
Mapping	Methods `doAcceptanceCheck` (implementation of check) and `isAcceptanceCheckOK` (outcome of check) in class `Telecommand`

FV8.2-4	**Telecommand Ready Check**
Description	The implementation of the ready check used in property P7.1.1-3 is application-specific.
Default	The default implementation of the ready check returns TRUE (telecommand is ready to start execution).
Range	This check must return either TRUE (telecommand is ready to start execution) or FALSE (telecommand is not yet ready to start execution).
Mapping	Methods `doReadyCheck` (implementation of check) and `isReadyCheckOK` (outcome of check) in class `Telecommand`

FV8.2-4	**Telecommand Start Check**
Description	The implementation of the start check used in property P7.1.1-4 is application-

	specific.
Default	The default implementation of the start check returns TRUE (telecommand execution can start successfully).
Range	This check must return either TRUE (telecommand execution can start) or FALSE (telecommand execution cannot be started).
Mapping	Methods `canStart` in class `Telecommand`

FV8.2-4	**Telecommand Progress Check**
Description	The implementation of the progress check used in properties P7.1.1-5 and P7.1.1-6 is application-specific.
Default	The default implementation of the progress check returns “telecommand has successfully completed its progress”.
Range	This check must return one of three values indicating: (1) telecommand execution is proceeding successfully but has not yet completed; (2) telecommand has successfully completed its progress; (3) telecommand cannot continue its execution.
Mapping	Methods `canProgress` (checks whether progress has been completed) and `hasProgressFailed` (check whether progress has failed) in class `Telecommand`

FV8.2-4	**Telecommand Completion Check**
Description	The implementation of the completion check used in property P7.1.1-7 is application-specific.
Default	The default implementation of the completion check returns TRUE (telecommand can successfully complete execution).
Range	This check must return either TRUE (telecommand completed execution successfully) or FALSE (telecommand did not complete execution successfully).
Mapping	Methods `canComplete` in class `Telecommand`

8.3 Factors of Variation for Telecommand Loading Concept

This section describes the mapping for the factors of variation associated to the telecommand loading concept.

FV8.3-1	**Telecommand Loading Action**
Description	The implementation of the telecommand loading action used in property P7.1.3-1 is application-specific.
Default	There is no default implementation for the telecommand loading action.
Range	This action must result in the framework telecommand component created by the telecommand loader being loaded in at least one telecommand manager.
Mapping	Method `loadIntoTcManager` in class `TelecommandLoader`

8.4 Factors of Variation for Telecommand Stream Concept

This section describes the mapping for the factors of variation associated to the telecommand stream concept.

FV8.4-1	**Telecommand Stream**
Description	The implementation of all the operations defined on the telecommand stream and used in property P7.1.3-1 is application-specific.
Default	There is no default implementation for the telecommand stream operations.
Range	Unrestricted.
Mapping	Interface `ITelecommandStream`

8.5 Factors of Variation for Telemetry Concept

This section describes the mapping for the factors of variation associated to the telemetry concept.

FV8.5-1	**Telemetry Configuration Action**
Description	The implementation of the telemetry configuration action used in property P7.2.1-1 is application-specific.
Default	The default implementation of this action returns without taking any action.
Range	Unrestricted.
Mapping	Method `doConfigureTm` in class `Telemetry`

FV8.5-2	**Telemetry Update Action**
Description	The implementation of the telemetry update action used in property P7.2.2-4 is application-specific.
Default	The default implementation of this action returns without taking any action.
Range	Unrestricted.
Mapping	Method `doUpdate` in class `Telemetry`

FV8.5-3	**Telemetry Serialization Action**
Description	The implementation of the telemetry update action used in property P7.2.2-4 is application-specific.
Default	There is no default implementation for this action.
Range	Unrestricted.
Mapping	Method `doSerialize` in class `Telemetry`

FV8.5-4	**Telemetry Termination Action**
Description	The implementation of the telemetry termination action used in property P7.2.2-7 is application-specific.

Default	The default implementation of this action returns without taking any action.
Range	Unrestricted.
Mapping	Method `doTerminate` in class `Telemetry`

FV8.5-5	**Telemetry Hold Check**
Description	The implementation of the telemetry hold check used in property P7.2.2-3 is application-specific.
Default	The default implementation of this check returns FALSE (packet is not held).
Range	This check must return either TRUE (telemetry packet is being held) or FALSE (telemetry packet is not being held).
Mapping	Method `isHeld` in class `Telemetry`

FV8.5-5	**Telemetry Termination Check**
Description	The implementation of the telemetry termination check used in property P7.2.2-6 is application-specific.
Default	The default implementation of this check returns TRUE (packet has terminated).
Range	This check must return either TRUE (telemetry packet has terminated) or FALSE (telemetry packet has not yet terminated).
Mapping	Method `canTerminate` in class `Telemetry`

8.6 Factors of Variation for Telemetry Stream Concept

This section describes the mapping for the factors of variation associated to the telemetry stream concept.

FV8.6-1	**Telemetry Stream**
Description	The implementation of all the operations defined on the telemetry stream and used in property P7.2.2-1 is application-specific.
Default	There is no default implementation for the telemetry stream operations.
Range	Unrestricted.
Mapping	Interface `ITelemetryStream`

9 CONTROL FRAMEWORK – UML2 MODEL ARCHITECTURE

This section describes the architecture of the UML2 model of the Control Framework. The framework architecture is described in terms of the main components that are defined by the framework UML2 model.

This section is intended to place the reader in a better position to understand the documentation attached to the UML2 design model.

9.1 Target Domain

The target domain of the Control Framework is defined in RD37. For the convenience of the reader, an outline of this domain is presented below.

The core of an AOCS application is the implementation of transfer functions that transform measurements from a set of sensors into commands for a set of actuators. Such transfer functions are implemented as digital filters that are characterized by a set of inputs, a set of outputs, an internal state, and an algorithm to compute the next set of outputs from the latest set of inputs and the current internal state.

Peripheral functionalities that are often found in AOCS applications are:

- Management of AOCS operational modes;
- Management of the external sensors and actuators;
- Implementation of failure detection and isolation checks and recovery actions (FDIR);
- Execution of AOCS-specific telecommands;
- Generation of AOCS-specific telemetry packets.

The Control Framework directly covers the management of the AOCS operational mode through the operation mode concept and the activity manager concept (see RD37 for an exhaustive discussion).

The Control Framework does not cover the management of the external sensors and actuators. This is due to the fact that the interfaces of AOCS sensors and actuators are neither standardized nor do they exhibit any significant commonalities in existing missions. Standardization of interfaces to external units (not just for the AOCS subsystem) is possible but this is done at the bus interface level, not at the functional level. Such functionalities are therefore not specific to the AOCS and are best left out of a framework targeted at the AOCS.

The FDIR functionalities are not directly included in the Control Framework in its present form. It is believed that such functionalities present sufficient commonalities to be implemented in reusable and adaptable component and they might be included in a future release of the Control Framework.

The management of the AOCS telecommands and AOCS telemetry is not directly included in the Control Framework. However, the DH Framework is interoperable with the Control Framework and hence these functionalities are supported by the two frameworks taken together.

9.2 Design Pattern Heritage

The definition of the Control Framework design uses the design patterns dealing with the implementation and management of *control blocks, data items, data pools,* and *parameter database* in the OBS Framework [RD18].

9.3 High-Level Functions

The components and interfaces defined by the Control Framework can be allocated to the following high-level functions:

- Activity Management Function: this function is responsible for the management of control activities. Note that control activities are organized in “activity sets” which are in turn controlled by “operational modes”. These concepts and their mutual relationships are described in greater detail in RD37.
- Initialization and Configuration Function: this function is responsible for the management of the initialization and configuration process of other framework components.

Each framework component or framework interface is allocated to one and only one function. The UML2 model of the Control Framework defines a dedicated package for each function as indicated in table 9.3.1.

The Initialization and Configuration Function is shared with the DH Framework. The same package is used by both frameworks. These two functions are therefore not described further in this section. The next section describes the Activity Management Function.

Table 9.3-1: Allocation of Functions to Packages

Function	Package
Activity Management Function	`Activity` Package
Initialization and Configuration Function	`Basic` Package

9.4 Activity Management Function

Figure 9.4-1 shows the architecture of the Activity Management Function in an informal notation. The boxes represent components or interfaces provided by the framework. The arrows represent flows of control or of data.

The core component is the activity component. An activity component encapsulates a control algorithm or some other sequential flow of actions to be executed by the control applications. Activities are characterized in the Control Framework by interface `IActivity`. The framework also defines a class implementing this interface and implementing the invariant behaviour of activities (class `Activity`).

Activities should normally only be manipulated by the activity manager and by the activity registry.

The activity manager component is responsible for executing activities. This component is characterized in the Control Framework by interface `IActivityManager`. The framework implements this component in full through three classes: `ActivityManager`, `OperationalMode`, and `ActivitySet`.

Class `ActivityManager` implements the logic for triggering changes in the internal states of an activity. Class `OperationalMode` implements a collection of activity sets representing an operational mode. Class `ActivitySet` implements a collection of activities representing an activity set. These three classes are intended to be accessed exclusively through the `IActivityManager` interface.

The Activity Registry component behaves as a *filter*. At every activation, before executing an activity, the activity manager passes the activity through the Activity Registry. This gives it the chance to change its enable and held status. This mechanism ensures that changes to the enable and held status of an activity are synchronized with the execution of an activity and that they occur at a well-defined point in the execution cycle of the activity.

Other components (either inside or outside the control part of an application) who wish to change the enable or held status of an activity, should do so by making a request to the activity registry. When such a request is made, the activity is identified by its name (namely by the *activity identifier*). The activity registry translates the activity name into a reference to the component implementing the activity and it buffers the request for the change in the activity status. The change is performed at the next execution cycle of the activity. Thus, the filter directly operates on the activity (by, for instance, changing its enable state).

The Activity Registry component is characterized in the Control Framework by interface `IActivityRegistry`. There is no default or partial implementation for this interface in the framework since no invariant behaviour can be associated to activity registries.

Applications may need to define activities that have other dynamically settable state properties in addition to the enable and held status. In such a case, the applications should extend interface `IActivityRegistry` to handle these additional aspects of the activity state.

In a control application, there can only be one single instance of Activity Registry.

In figure 9.4-1, the data pool and parameter database components are shown in dashed boxes to represent the fact that they are not defined in the UML2 model of the Control Framework. These two components are defined by a meta-model that is documented in section 14.

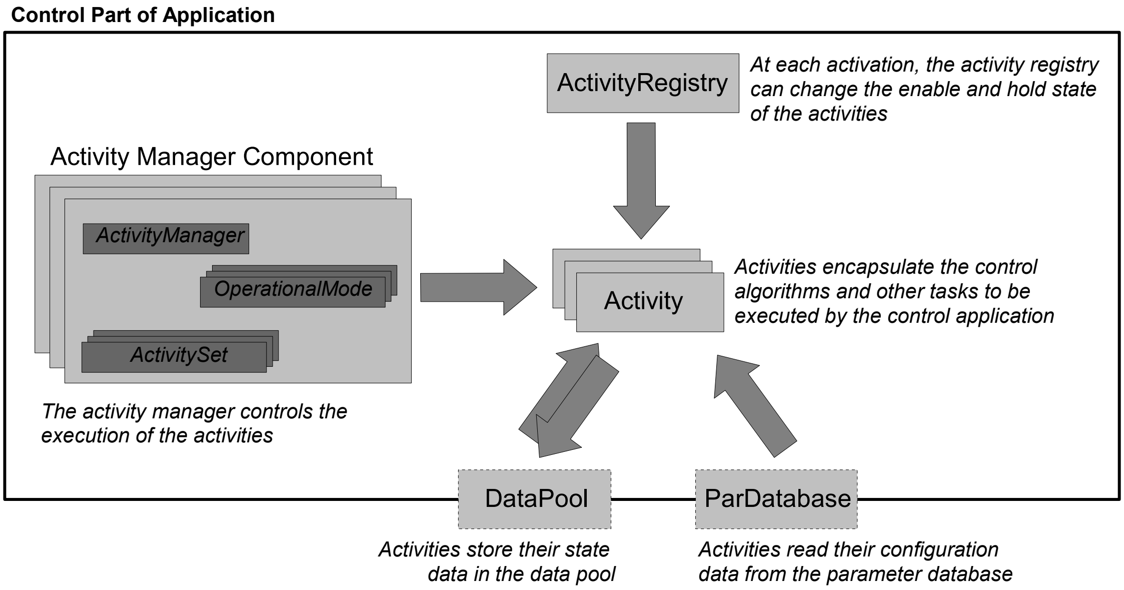

Control Part of Application
ActivityRegistry
At each activation, the activity registry can change the enable and hold state of the activities
Activity Manager Component
ActivityManager
OperationalMode
ActivitySet
Activity
Activities encapsulate the control algorithms and other tasks to be executed by the control application
The activity manager controls the execution of the activities
DataPool
ParDatabase
Activities store their state data in the data pool
Activities read their configuration data from the parameter database

Fig. 9.4-1: Architecture of Activity Management Function

10 CONTROL FRAMEWORK – DOMAIN DICTIONARY MAPPING

This section describes how the domain dictionary entries for the Control Framework are mapped to elements in the UML2 design model of the framework.

The format in which the mapping is described is the same as for the DH Framework domain dictionary entry (see section 6). The general considerations made in the introduction to section 6 for the DH Framework also apply to the Control Framework.

10.1 Mapping of Domain Dictionary Entries for Activity Concept

This section defines the mapping to design elements for the domain dictionary entries related to the activity concept.

Note that the operations, actions, and checks relative to the activity initialization are mapped to methods in the base class `Component`. In other words, the activity initialization properties and factors of variation are implemented at the level of the `Component` super-class.

Dictionary Term	**Activity**	**Interface `IActivity`**
Attributes	Type	`activityType` attribute in `Activity`
	Subtype	`activitySubType` attribute in `Activity`
	Identifier	`activityID` attribute in `Activity`
Operations	Initialize	`initialize`, and `reset` methods in `IComponent`
	Execute	`ReadInputs, writeOuptut`, and `Propagate` methods in `IActivity`
	Enable	`Enable` method in `IActivity`
	Disable	`Disable` method in `IActivity`
	Hold	`Hold` method in `IActivity`
	Resume	`Release` method in `IActivity`
Actions	Initialization Action	`doInitialize` and `doConfigure` methods in `IComponent`
	Input Read Action	`doReadInput` method in `Activity`
	Output Write Action	`doWriteInput` method in `Activity`
	Propagation Action	`doPropagate` method in `Activity`
	Start Action	`doStart` method in `Activity`
	End Action	`doEnd` method in `Activity`
Checks	Initialization Check	`canInitialize` and `canConfigure` methods in `IComponent`
	Propagation Check	`canPropagate` method in `Activity`

10.2 Mapping of Domain Dictionary Entries for Mode Management Concept

This section defines the mapping to design elements for the domain dictionary entries related to the mode management concept.

Mode management is encapsulated in one single component characterized by interface `IActivityManager`. This component consists of three classes: `ActvityManager`, `OperationalMode`, and `ActivitySet`. The mode management concept is mapped to all three classes.

Dictionary Term	**Operational Mode**	**Classes Operational Mode and ActivitySet**
Attributes	Identifier	`operationalModeID` attribute in `OperationalMode`
	Max number of activity sets	`ActivitySetCounts` attribute in `OperationalMode`
	Max number of activities in each set	`activityCount` attribute in `ActivitySet`
Operations	Query for list of activities	`getActivitySets` method in `OperationalMode`
	Load an Activity	`load` method in `OperationalMode`
	Unload an Activity	`unload` method in `OperationalMode`
Actions	Mode Entry Action	`modeEntryAction` method in `OperationalMode`
	Mode Exit Action	`modeExitAction` method in `OperationalMode`
Checks	Mode Entry Check	`canEnter` method in `OperationalMode`
	Mode Exit Check	`canExit` method in `OperationalMode`

Dictionary Term	**Activity Manager**	**Class ActivityManager**
Attributes	Identifier	`activityManagerID` attribute in `ActivityManager`
	Current Mode	`currentMode` attribute in `ActivityManager`
Operations	Activate	`activate` method in `ActivityManager`
	Request mode transition	`changeMode` method in `ActivityManager`
Actions	Mode Update Action	`modeUpdateAction` method in `ActivityManager`
Checks	Mode Update Check	`isModeChangeNeeded` method in `ActivityManager`
	Mode Transition Check	`canChangeMode` method in `OperationalMode`

10.3 Domain Dictionary Entries for Parameter Database Concept

Unlike the activity and mode management concepts, the parameter database concept is not mapped to the UML2 part of the Control Framework. It is instead mapped to the parameter database meta-model which is discussed in section 14.

10.4 Domain Dictionary Entries for Data Pool Concept

Unlike the activity and mode management concepts, the parameter database concept is not mapped to the UML2 part of the Control Framework. It is instead mapped to the parameter database meta-model which is discussed in section 14.

11 CONTROL FRAMEWORK – MAPPING OF SHARED PROPERTIES

This section describes how the shared properties defined at domain analysis level for the Control Framework have been mapped to the UML2 model of the framework. The formal with which the mapping is described is the same as used for the DH Framework and as described in the introduction to section 7.

11.1 Shared Properties for Activity Concept

This section describes the mapping of the shared properties associated to the activity concept of the Control Framework.

11.1.1 Activity Initialization Properties

The properties relative to the activity initialization are implemented in the state machine of class `Component`. This state machine is the parent state machine of all activities. The activity state machine is embedded within state CONFIGURED in the `Component` state machine.

P11.1.1-1	*The initialization operation on an activity is only successful if the initialization check returns TRUE. If this is not the case, then the initialization operation on the activity has no effect.*
	(1) State INITIALIZED in the state machine associated to class `Component` can only be entered if method `canInitialize` in the same class returns TRUE when the transition into the state is attempted. *(2) State CONFIGURED in the state machine associated to class `Component` can only be entered if method `canConfigure` in the same class returns TRUE when the transition into the state is attempted.*
P11.1.1-2	*If the initialization operation on an activity is successful, then the activity performs its initialization action.*
	When state INITIALIZED in the state machine associated to class `Component` is entered, then method `doInitialize` is executed. *When state CONFIGURED in the state machine associated to class `Component` is entered, then method `doConfigure` is executed.*
P11.1.1-3	*Unless an activity has been successfully initialized, all other operations performed upon it have no effect.*
	The trigger methods defined by class `Activity` trigger a state transition only when the state machine associated to the class is in state CONFIGURED.

11.1.2 Activity Execution

The properties relative to the activity execution are implemented in the state machine of class `Activity`. The execution of an activity is performed by an activity manager and it consists in the execution of three methods in sequence: `readInputs`, `propagate`, and `writeOutputs`.

P11.1.2-1	*When an activity that is neither disabled nor held is executed, it performs its propagation check and, if this returns TRUE, then the activity performs its input read operation, its propagation action , and its output write operation.*
	Pre-condition: the activity is loaded in the ActivityManager. *If method `activate` is called on the activity manager, and if the call does not result in a change in operational mode, and if the activity is in state ACTIVE, then the activity goes through states EXECUTE1, EXECUTE2, and then returns to state ACTIVE.*

11.1.3 Holding and Resuming Activities

The properties relative to the activity execution are implemented in the state machine of class `Activity`. The execution of an activity is performed by an activity manager and it consists in the execution of three methods in sequence: `readInputs`, `propagate`, and `writeOutputs`.

P11.1.3-1	*An execute operation performed upon an activity that is held has no effect.*
	Pre-condition: the activity is loaded in the ActivityManager. *If method `activate` is called on the activity manager, and if the call does not result in a change in operational mode, and if the activity is in state HELD, then the activity remains in this state.*
P11.1.3-2	*If a resume operation is performed upon an activity that is held, then the activity is no longer held (ie it is resumed).*
	If method `resume` is called on an activity that is in state HELD, then the activity makes the transition to state ACTIVE.
P11.1.3-3	*If a hold operation is performed upon an activity that is not held, then the activity is held.*
	If method `held` is called on an activity that is in state ACTIVE, then the activity makes the transition to state HELD.

11.1.4 Enabling and Disabling of Activities

The properties relative to the activity execution are implemented in the state machine of class `Activity`. The execution of an activity is performed by an activity manager and it consists in the execution of three methods in sequence: `readInputs`, `propagate`, and `writeOutputs`.

P11.1.4-1	*When an activity is disabled, it performs its end action.*
	When an activity enters state DISABLED, then it executes method `doEndAction`.
P11.1.4-2	*When an activity is enabled, it performs its start action.*
	When an activity leaves state DISABLED, then it executes method `doStartAction`.
P11.1.4-3	*An execute operation performed on an activity that is disabled has no effect.*
	Pre-condition: the activity is loaded in the ActivityManager . *If method `activate` is called on the activity manager, and if the call does not result in a change in operational mode, and if the activity is in state DISABLED, then the activity remains in this state.*
P11.1.4-4	*If a disable operation is performed upon an activity that is enabled, then the activity becomes disabled.*
	If method `disable` is called on an activity that is in state ACTIVE, then the activity makes the transition to state DISABLED.
P11.1.4-5	*If an enabled operation is performed upon an activity that is disabled, then the activity becomes enabled.*
	If method `enable` is called on an activity that is in state DISABLED, then the activity makes the transition to state ACTIVE.

11.2 Shared Properties for Mode Management Concept

This section describes the mapping of the shared properties associated to the operational mode and activity manager concepts.

11.2.1 Activity Manager Activation

The properties relative to the activity manager activation are implemented in the methods of class `ActivityManager`. The implementation of these methods is modelled using the action language of the FW Profile.

P11.2.1-1	*When an activity manager is activated, it first performs the mode update check. It then performs a change in current operational mode (if this is required by the outcome of the mode update check), and it finally executes all the activities in the current operational mode.*
	When method activate is called on an activity manager, then method `isModeChangeNeeded` is called. If this returns true, then method `changeMode` is called with its argument equal to the return value of method `getTargetMode`. Finally, method `activate` is called on all activities in the current operational mode.

11.2.2 Current Operational Mode Changes

The current operational mode of an activity manager can change either as a result of a request from some outside entity (mode transition request) or as a result of an autonomous decision of the activity manager itself (mode update check).

In both cases, the change in current mode can only take place under certain conditions. More specifically, there are three types of checks that are performed by an operational mode:

- The *mode exit check* verifies that the current operational mode can be exited.
- The *mode entry check* verifies that the target operational mode can be entered.
- The *mode transition check* verifies that the transition from the current to the target operational mode can be performed (ie it verifies the legality of the transition across two modes)

The change in operational mode is only performed if all three checks are positive. When an activity manager changes its operational mode, then it executes its mode update action.

When an operational mode is phased out from being the current mode, then the activity manager executes the end actions associated to all its activities. Similarly, when an operational mode is phased in as new current mode, the activity manager executes all its start actions.

Note that the order in which the start and end actions of the activities in an operational mode are executed is undefined because the operational mode is a *set* of activities (as opposed to an *ordered list*). The start and end actions of activities are factors of variation of the Control Framework and their content is defined by the application designer during the framework instantiation process. However, in order to preserve determinism of behaviour, they are restricted to modify only the internal state of the activity itself (see section 12.2).

Changes to the global state of the framework (namely to the content of the data pool) that should take place when an operational mode changes, must be encapsulated in the entry and exit actions of the operational mode themselves.

P11.2.2-1	*A change in current operational mode from a source to a destination operational mode is only performed if the mode exit check of the source mode, the mode entry check of the destination mode, and the mode transition check on the [source,destination] pair are successful.*
	A call to method `activate` on `ActivityManager` can only result in a change in current operational mode if all the following conditions hold: method `canExit`

	in the source mode returns TRUE, method `canEnter` in the destination mode returns TRUE, and method `canChangeMode` on the [source, destination] mode returns TRUE.
P11.2.2-2	*When an operational mode is phased out from being the current mode of an activity manager, the activity manager executes the end action of all its activities.*
	If a call to method `activate` on `ActivityManager` results in a change in current operational mode, then method `end` is executed on all activities of the source mode and this results in method `doEndAction` being executed on them.
P11.2.2-3	*When an operational mode is phased in as new current mode of an activity manager, the activity manager executes the start action of all its activities.*
	If a call to method `activate` on `ActivityManager` results in a change in current operational mode, then method `start` is executed on all activities of the source mode and this results in method `doStartAction` being executed on them.
P11.2.2-4	*When an activity manager changes its current mode, then it executes its mode update action.*
	If a call to method `activate` on `ActivityManager` results in a change in current operational mode, then its method `modeUpdateAction` is executed.
P11.2.2-5	*When an operational mode becomes the new current mode, it executes its mode entry action.*
	If a call to method `activate` on `ActivityManager` results in a change in current operational mode, then method `modeEntryAction` is executed on the new operational mode.
P11.2.2-6	*When an operational mode is phased out from being the current mode, it executes its mode exit action.*
	If a call to method `activate` on `ActivityManager` results in a change in current operational mode, then method `modeExitAction` is executed on the old operational mode.

11.3 Shared Properties for Parameter Database Concept

Unlike the activity and mode management concepts, the parameter database concept is not mapped to the UML2 part of the Control Framework. It is instead mapped to the parameter database meta-model which is discussed in section 14. The shared properties associated to it are discussed in this section.

11.4 Shared Properties for Data Pool Concept

Unlike the activity and mode management concepts, the data pool concept is not mapped to the UML2 part of the Control Framework. It is instead mapped to the data pool meta-model which is discussed in section 14. The shared properties associated to it are discussed in this section.

12 CONTROL FRAMEWORK – FACTORS OF VARIATION

This section describes how the factors of variation defined at domain analysis level for the Control Framework are mapped to the UML2 design model of the framework. The format of the description of the mapping is the same as described in the introduction to section 8. The general considerations made in that section for the DH Framework also apply to the Control Framework.

12.1 Attributes as Factors of Variation

All the attributes attached to framework classes represent implicit factors of variation since attribute values are intended to be defined by the application designer at framework instantiation time.

Factors of variations linked to attributes are regarded as trivial and implicitly defined by the domain model and are therefore not further considered in this section.

12.2 Factors of Variation for Activity Concept

This section describes the mapping of the factors of variation associated to the activity concept.

With respect to the initialization check and actions, it should be recalled that activity initialization is mapped to the generic initialization and configuration mechanism offered by the `Component` super-class.

FV12.2-1	**Activity Initialization Check**
Description	The implementation of the initialization check used in property P11.1.1-1 is application-specific.
Default	The default implementation of the initialization check returns TRUE if all the attributes of the activity have legal values.
Range	This check must return either TRUE (activity has been successfully initialized) or FALSE (activity has not been successfully initialized).
Mapping	Method `canInitialize` and `canConfigure` in class `Component`

FV12.2-2	**Activity Initialization Action**
Description	The implementation of the initialization action used in property P11.1.1-2 is application-specific.
Default	The default implementation of the initialization action does nothing.
Range	Unrestricted.
Mapping	Method `doInitialize` and `doConfigure` in class `Component`

FV12.2-3	**Activity Propagation Check**
Description	The implementation of the propagation check used in property P11.1.2-1 is application-specific.
Default	The default implementation of the propagation check returns TRUE.
Range	This check must return either TRUE (activity can be propagated) or FALSE

	(activity cannot be propagated).
Mapping	Method `canPropagate` in class `Activity`

FV12.2-4	**Activity Propagation Action**
Description	The implementation of the propagation action used in property P11.1.2-1 is application-specific.
Default	There is no default implementation for the propagation action.
Range	Unrestricted.
Mapping	Method `doPropagate` in class `Activity`

FV12.2-5	**Activity Input Read Action**
Description	The implementation of the input read action used in property P11.1.2-1 is application-specific.
Default	There is no default implementation for the input read action.
Range	Unrestricted.
Mapping	Method `doReadInputs` in class `Activity`

FV12.2-6	**Activity Output Write Action**
Description	The implementation of the output write action used in property P11.1.2-1 is application-specific.
Default	There is no default implementation for the output write action.
Range	Unrestricted.
Mapping	Method `doWriteOutputs` in class `Activity`

FV12.2-7	**Activity Start Action**
Description	The implementation of the start action used in property P11.1.4-2 is application-specific.
Default	The default implementation of the start action does nothing.
Range	This action can only affect the internal state of the activity (ie it cannot change the value of an entry in a data pool). See discussion in section 11.2.2.
Mapping	Method `doStartAction` in class `Activity`

FV12.2-8	**Activity End Action**
Description	The implementation of the end action used in property P11.1.4-1 is application-specific.
Default	The default implementation of the end action does nothing.
Range	This action can only affect the internal state of the activity (ie it cannot change the value of an entry in a data pool). See discussion in section 11.2.2.

Mapping	Method `doEndAction` in class `Activity`

12.3 Factors of Variation for Mode Management Concept

This section describes the mapping of the factors of variation associated to the mode management concept.

FV12.3-1	**Mode Update Check**
Description	The implementation of the mode update check used in property P11.2.1-1 is application-specific.
Default	The default implementation of the mode update check returns: 'no mode update required'.
Range	This check must return either TRUE (mode update is required) or FALSE (no mode update is required).
Mapping	Method `isModeChangeNeeded` in class `ActivityManager`

FV12.3-2	**Mode Exit Check**
Description	The implementation of the mode exit check used in property P11.2.2-1 is application-specific.
Default	The default implementation of the mode exit check returns: 'mode exit allowed'.
Range	This check must return either TRUE (mode exit is allowed) or FALSE (mode exit is not allowed).
Mapping	Method `canExit` in class `OperationalMode`

FV12.3-3	**Mode Entry Check**
Description	The implementation of the mode entry check used in property P11.2.2-1 is application-specific.
Default	The default implementation of the mode entry check returns: 'mode entry allowed'.
Range	This check must return either TRUE (mode entry is allowed) or FALSE (mode entry is not allowed).
Mapping	Method `canEnter` in class `OperationalMode`

FV12.3-4	**Mode Transition Check**
Description	The implementation of the mode transition check used in property P11.2.2-1 is application-specific.
Default	The default implementation of the mode transition check returns: 'mode transition allowed'.
Range	This check must return either TRUE (mode transition is allowed) or FALSE (mode transition is not allowed).
Mapping	Method `canChangeMode` in class `ActivityManager`

FV12.3-5	**Mode Update Action**
Description	The implementation of the mode update action used in property P11.2.2-4 is application-specific.
Default	The default implementation of the mode update action returns without doing anything.
Range	Unrestricted.
Mapping	Method `ModeUpdateAction` in class `ActivityManager`

FV12.3-2	**Mode Exit Action**
Description	The implementation of the mode exit action used in property P11.2.2-6 is application-specific.
Default	The default implementation of this action returns without doing anything.
Range	Unrestricted.
Mapping	Method `ModeExitAction` in class `OperationalMode`

FV12.3-3	**Mode Entry Action**
Description	The implementation of the mode entry action used in property P11.2.2-5 is application-specific.
Default	The default implementation of this action returns without doing anything.
Range	Unrestricted.
Mapping	Method `ModeEbtryAction` in class `OperationalMode`

12.4 Factors of Variation for Data Pool Concept

Unlike the activity and mode management concepts, the data pool concept is not mapped to the UML2 part of the Control Framework. It is instead mapped to the parameter database meta-model which is discussed in section 14. The factors of variation associated to it are discussed in that section too.

13 CONTROL FRAMEWORK – ACTIVITY EXAMPLES

The activity concept is perhaps the most novel among the concepts proposed by the CORDET Framework. It is therefore useful to illustrate its use and versatility through two examples. The example show how the activity concept can be extended to represent other concepts of common usage within the control domain.

13.1 The HealthCheck Activity

The need to perform health check is very common within the control domain in general and the AOCS domain in particular.

Health checks are often defined to periodically monitor some variable of interest and to check whether the behaviour of the variable indicates an anomaly. If an anomaly is detected, then the variable is declared to be 'suspected'. If the variable remains in this state more than a certain number of consecutive monitoring cycle, then a failure is declared.

This behaviour can be encapsulated in a component that extends the generic activity component offered by the Control Framework. Figure 13.1-1 shows the corresponding state machine using an informal notation. This state machine should be embedded within the state machine of the EXECUTE_2 state in the `Activity` component.

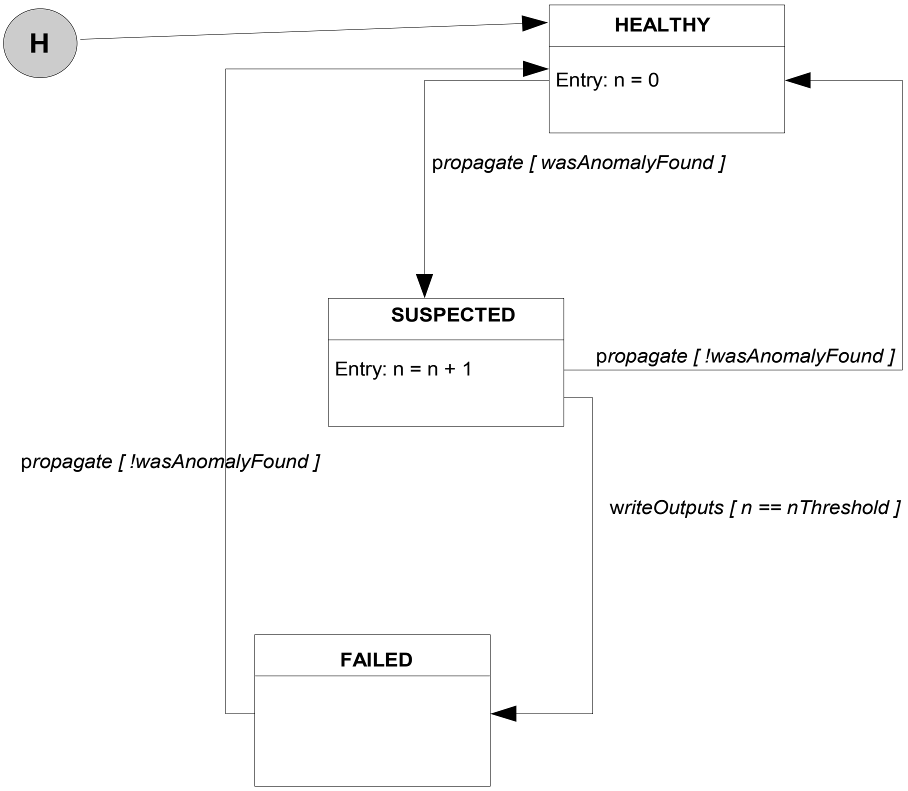

H
HEALTHY
Entry: n = 0
propagate [wasAnomalyFound]
SUSPECTED
Entry: n = n + 1
propagate [!wasAnomalyFound]
propagate [!wasAnomalyFound]
writeOutputs [n == nThreshold]
FAILED

Fig. 13.1-1: State Machine of HealthCheck Activity (Informal Notation)

The state machine basically works as follows:

- The health check is encapsulated in method `doPropagation` (which is defined in the base component `Activity` and must be overridden by the `HealthCheck` component)
- Method wasAnomalyFound reports the outcome of the check.

- Every time an anomaly is found, a counter n is incremented.
- A failure is declared if nThreshold consecutive anomalies are found.
- A declaration of failure is reversible.
- If the activity is disabled or ended, then the health check is reset and its state goes back to HEALTHY.
- Method `doEndAction` is defined in the base component `Activity` and must be overridden to reset the history state.

Application designers can further extend this activity to define their own specific health-checks. The advantage for them is that they inherit the general health check logic: they only have to define the specific anomaly detection checks.

Note that it is possible to define other types of 'reasonable' health check logic. Possible differences from the logic shown in the previous figure are:

- A declaration of failure could be made irreversible;
- A failed check where the anomaly disappears becomes SUSPECTED before becoming again HEALTHY
- Disabling and re-enabling the health check does not reset the health check
- Execution of a recovery action is embedded in state FAILED

Still other solutions are possible. The fact that no general health check logic can be define implies that the definition of health checks cannot really be done at the level of the Control Framework but is best left to lower levels of design.

13.2 The Manoeuvre Activity

The execution of manoeuvres (sequences of actions to achieve some high-level goal such as slewing the spacecraft or switching on or off a complex unit) are another common task in control systems in general and in AOCS in particular. For this purpose, too, it may be useful to define a dedicated component as an extension of the generic `Activity` component.

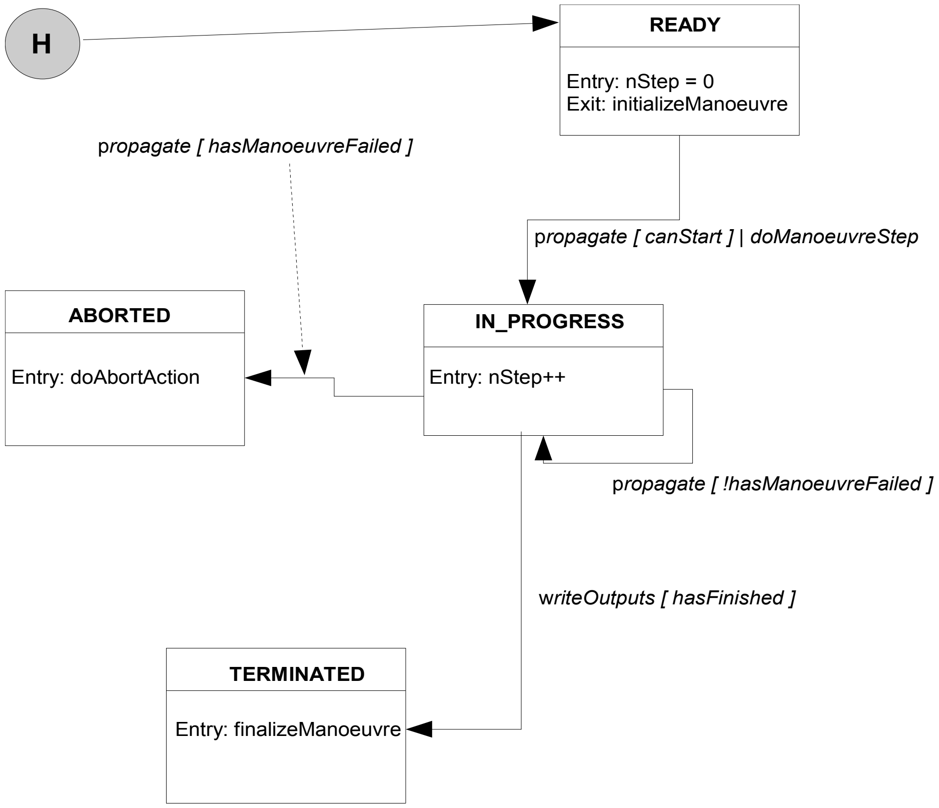

H
READY
Entry: nStep = 0
Exit: initializeManoeuvre
propagate [hasManoeuvreFailed]
propagate [canStart] | doManoeuvreStep
ABORTED
Entry: doAbortAction
IN_PROGRESS
Entry: nStep++
propagate [!hasManoeuvreFailed]
writeOutputs [hasFinished]
TERMINATED
Entry: finalizeManoeuvre

Fig. 13.2-1: State Machine of Manoeuvre Activity (Informal Notation)

The behaviour of a generic manoeuvre can be summarized as follows:

- The manoeuvre starts when its “start check” authorizes it.
- The manoeuvre executes in a number of steps.
- The manoeuvre can abort itself.
- A manoeuvre that is disabled, is reset.
- A manoeuvre can only be executed once before being reset.
- Method `doEndAction` must be overridden to reset the history state of the manoeuvre.

This component therefore encapsulates a generic manoeuvre management logic while at the same time offering adaptation points where application-specific behaviour can be added.

Again, as in the case of the health check component, it would be possible to define different – but still reasonable and useful – manoeuvre management logic. Possible alternative choices are:

- Manoeuvres that can be executed more than once without being reset
- Manoeuvres that are capable of undoing one or more of its steps
- Manoeuvres that have explicit branching logic
- Manoeuvres that have a fixed number of steps

The fact that different logic can be define within the same control domain suggests that this type of component exists at a level of abstraction lower than the level at which the more general activity or operational mode concepts are defined.

14 DATA POOL AND PARAMETER DATABASE META-MODEL

There are three reasons why the data pool and parameter database concepts were not mapped to the UML2 model used for the rest of the Control Framework:

1. UML2 is well-suited to modelling software entities with a fixed structure. Functional variability is modelled by leaving 'holes' in the structure under the guise of virtual or pure virtual methods. The variability associated to the data pool and parameter database concepts tends to be structural: it concerns the mode of access to the data items and parameter values and the structure of the information behind each single data item.
2. The designers of on-board applications normally represent the on-board data using spreadsheets or database that describe the structure and content of each data item.
3. One implication of the activity concept proposed by the Control Framework is that access to the data pool is likely to constitute a performance bottleneck for applications instantiated from the framework. Language- and platform-specific optimizations may therefore be necessary when defining both the design and the implementation of the data pool. To a lesser extent, this may also apply to the parameter database design and implementation. A design based on a meta-model that only capture the essential structural features of the data pool and parameter database is preferable because it facilitates the optimization during application development.

In principle, the difficulty mentioned at point 1 could be overcome by having a very complex UML2-based model of a data pool or parameter database interface. Such an interface would have to cover all possible structural variants of the data pool and database concept. An implementation would only use a subset of the operations defined on this interface.

This approach would be complex but seems over-complex. Instead an alternative approach was selected which allowed to remain closer to current practice. In the selected approach, at design level, a *meta-model* is defined for the data pool and parameter database concepts. The meta-model is formally expressed as a feature model but this is then mapped to a spreadsheet with a fixed structure.

Note that the other parts of the Control Framework do not interact with the data pool and parameter database concepts. This is makes it possible to select a different representation for their design models.

Activities constitute the link between the part of the Control Framework modelled with UML2 and the data pool and parameter database parts of the framework. The link with the data pool is encapsulated in methods `readInputs` and `writeOutputs` in the `Activity` component. These are pure virtual methods which are likely to be instantiated only at the lowest level of abstraction (namely on the final classes created during the framework instantiation process). This means that the decoupling between the data pool model and the UML2 model of the framework classes can be maintained until the end of the application design process.

The Control Framework does not model the link between activities and the parameter database but application designer could use an approach similar to that used for the data pool with the same objective of maintaining the decoupling with the parameter database model until the end of the application design process.

The data pool and parameter database meta-models are conceptually separate but, in practice and for the sake of simplicity, they are implemented as one single feature model.

Sections 14-1 and 14-2 describe the data pool and parameter database meta-models, respectively. These meta-models are implemented as feature models. The following subsection describes how these feature models can be mapped to the *framework configuration file*.

14.1 Data Pool Meta-Model

The data pool meta-model is expressed as a feature model. The feature model is defined using the XFeature tool[6] with the "FD Configuration". In other words, the same tool with the same configuration is used for the data pool meta-model as was used for the domain model feature models described in reference RD-37.

The feature model representing the data pool meta-model is shown in figure 14.1-1.

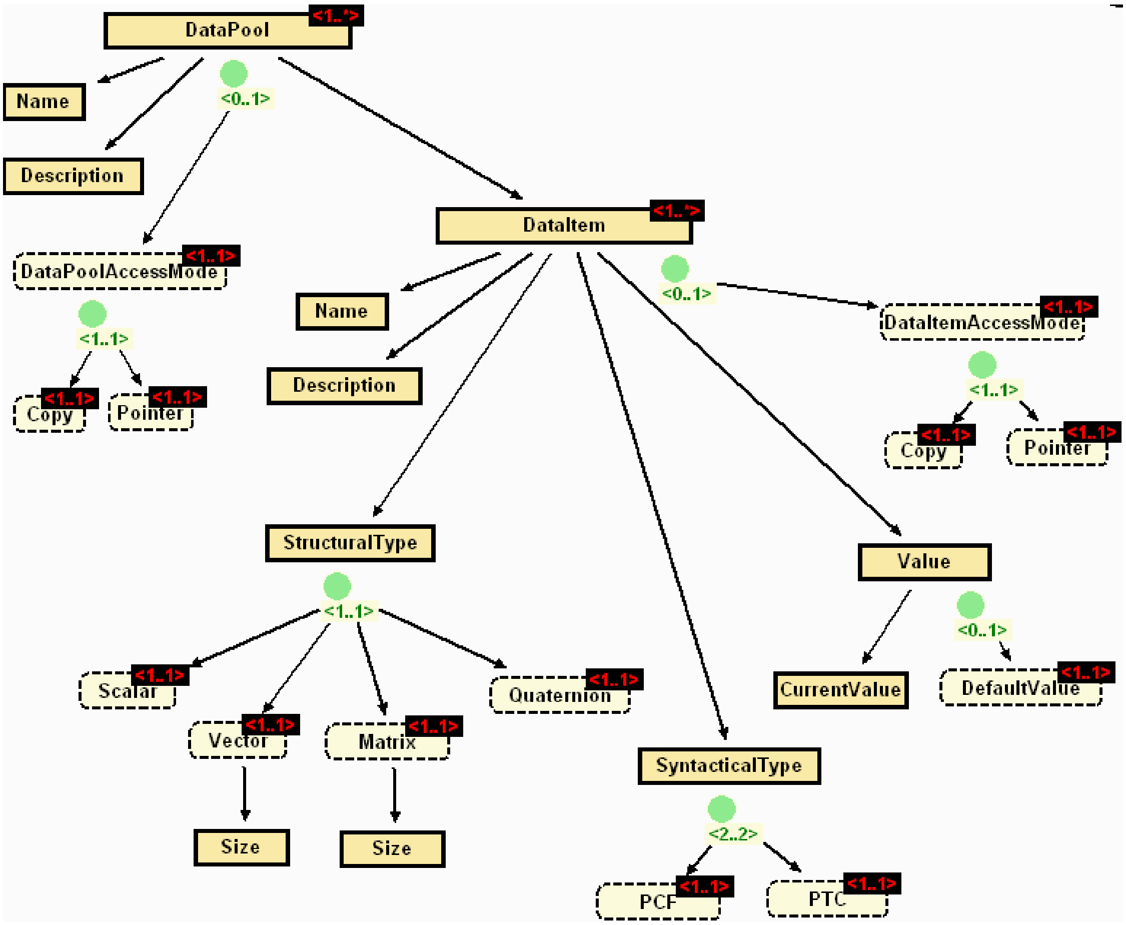


Fig. 14.1-1: Feature Model for the Data Pool Meta-Model

The following clarifications are in order with reference to the feature model of figure 14.1-1:

- There may be several data pools in the same applications. Each data pool has a name (the data pool identifier) and a description. Each data pool holds a number of data items.
- There are two modes of access to data pool items: either by *copy* or by *pointer*. It is possible either to specify a mode of access at data pool level that applies to all items in the data pool, or it is possible to specify a mode of access at data item level that applies to each individual data item. Note that the feature model of the data pool meta-

[6] The tool can be downloaded as free and open software from its home page at this address: http://www.pnp-software.com/XFeature/

model defines an XFeature global constraint (which is, of course, not visible in figure 14.1-1) to express the fact that, if the mode of access is defined at data pool level, then it cannot be defined at data item level.

- Each data item has a name (its identifier) and a description. The name should be unique. Note that this constraint cannot be expressed in the feature model. .
- Data items have a structural type which may be: scalar, vector, matrix, or quaternion. For matrices and vectors, the size must be specified too.
- The syntactical type of data items is expressed using the PTC and PFC fields defined by the PUS.
- To each data pool item, a default value may be associated.

14.2 Parameter Database Meta-Model

The parameter database meta-model is expressed as a feature model. The feature model is defined using the XFeature tool[7] with the “FD Configuration”. In other words, the same tool with the same configuration is used for the parameter database meta-model as was used for the domain model feature models described in reference RD-37.

In practice, the feature model for the parameter database meta-model is embedded within the same feature model that hosts the data pool meta-model.

The feature model representing the data pool meta-model is shown in figure 14.2-1.

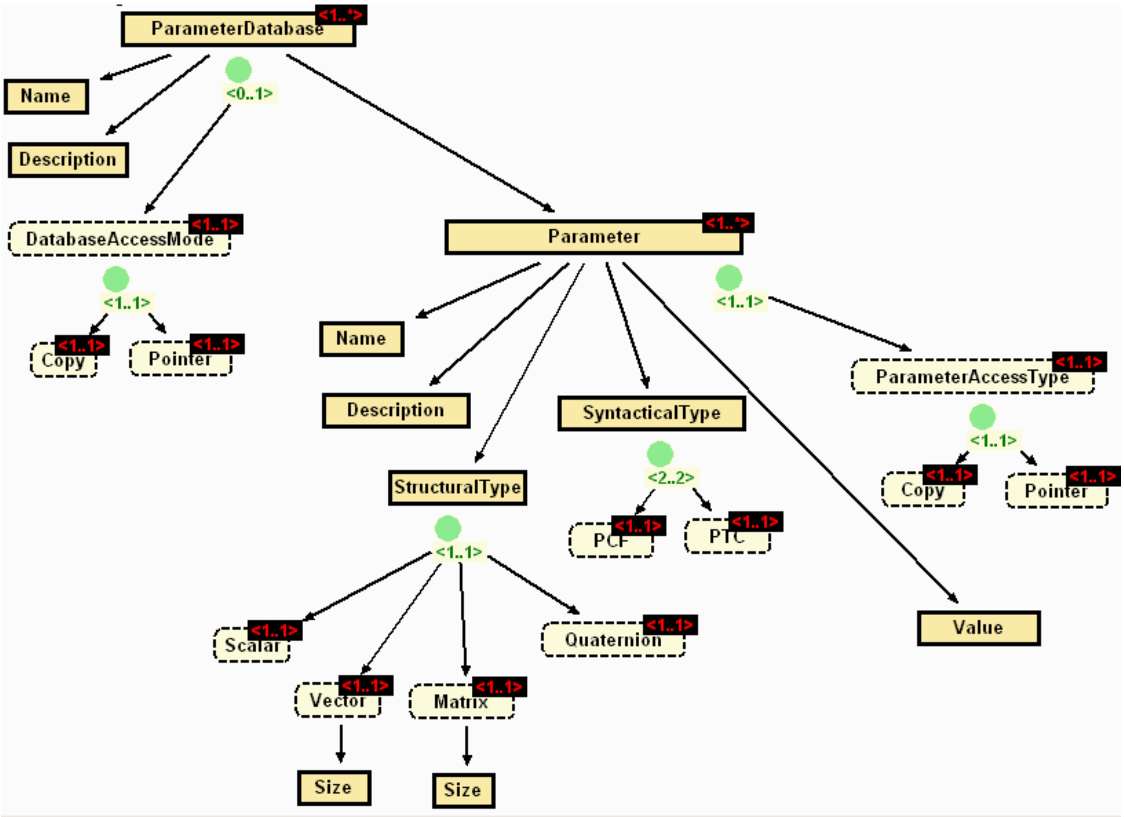


Fig. 14.2-1: Feature Model for the Parameter Database Meta-Model

[7] The tool can be downloaded as free and open software from its home page at this address: http://www.pnp-software.com/XFeature/

The following clarifications are in order with reference to the feature model of figure 14.2-1:

- There may be several parameter databases in the same applications. Each parameter database has a name (the parameter database identifier) and a description. Each parameter database holds a number of parameters.
- There are two modes of access to parameters: either by *copy* or by *pointer*. It is possible either to specify a mode of access at parameter database level that applies to all items in the parameter database, or it is possible to specify a mode of access at parameter level that applies to each individual parameter. Note that the feature model of the data pool meta-model defines an XFeature global constraint (which is, of course, not visible in figure 14.2-1) to express the fact that, if the mode of access is defined at parameter database level, then it cannot be defined at parameter level.
- Each parameter has a name (its identifier) and a description. The name should be unique. Note that this constraint cannot be expressed in the feature model. .
- Parameters have a structural type which may be: scalar, vector, matrix, or quaternion. For matrices and vectors, the size must be specified too.
- The syntactical type of parameters is expressed using the PTC and PFC fields defined by the PUS.
- To each parameter, a value must be associated.

14.3 Framework Configuration File

The meta-models in the previous two sections have been defined as family-level models within the XFeature graphical environment. The advantage of this choice is that XFeature family-level models can be automatically transformed into application meta-models. The meta-models are implemented as XSD schemas.

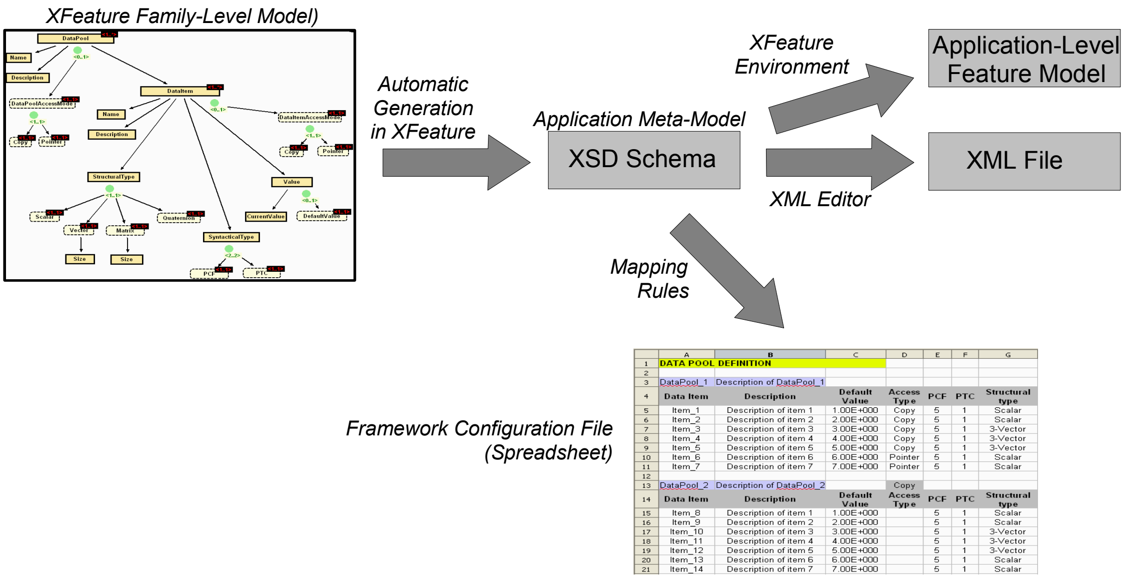
These XSD-based meta-models can be used by application designer to define their specific and concrete data pool and parameter database models. The definition can either be done within the XFeature environment or in any XML editor. In the former case, the application-level models are built as feature models within the XFeature graphical environment. The graphical representation is, of course, automatically serialized to an XML-file that complies with the XSD schema. In the latter case, they are built as text-based XML files that co,ply with the XSD schema.

The two options are ultimately equivalent (they both result in the same XML-based representation) and both are possible. In practice, however, both conflict with current practice and would be awkward to use with the typical size of on-board data pools or parameter database. As already mentioned at the beginning of this section, current practice is based is based on the use of spreadsheets. This is convenient primarily because on-board data pool and parameter databases will often have thousands of entries and hence a purely graphical representation (XFeature solution) or a purely XML solution would be impractical.

Hence, in the CORDET project, a mixed solution is adopted where the data pool and parameter database meta-models are formally defined as XFeature family-level feature models but these are then not transformed into an XSD-based feature meta-model. Instead, they are mapped to a spreadsheet and the spreadsheet is used during framework instantiation to define the data pool and parameter database for a specific and concrete application.

The spreadsheet to which the data pool and parameter database meta-models are mapped is called *Framework Configuration File*. This name has been chosen because this same file can also be used to describe other aspects of a framework instantiation in addition to the data pool and parameter database configuration.

Figure 14.3-1 illustrates the previous discussion by showing the way in which a family-level feature model representing the variability within a framework can be transformed into application level configuration information.



	A	B	C	D	E	F	G
1	DATA POOL DEFINITION						
2							
3	DataPool_1	Description of DataPool_1					
4	Data Item	Description	Default Value	Access Type	PCF	PTC	Structural type
5	Item_1	Description of item 1	1.00E+000	Copy	5	1	Scalar
6	Item_2	Description of item 2	2.00E+000	Copy	5	1	Scalar
7	Item_3	Description of item 3	3.00E+000	Copy	5	1	3-Vector
8	Item_4	Description of item 4	4.00E+000	Copy	5	1	3-Vector
9	Item_5	Description of item 5	5.00E+000	Copy	5	1	3-Vector
10	Item_6	Description of item 6	6.00E+000	Pointer	5	1	Scalar
11	Item_7	Description of item 7	7.00E+000	Pointer	5	1	Scalar
12							
13	DataPool_2	Description of DataPool_2		Copy			
14	Data Item	Description	Default Value	Access Type	PCF	PTC	Structural type
15	Item_8	Description of item 1	1.00E+000		5	1	Scalar
16	Item_9	Description of item 2	2.00E+000		5	1	Scalar
17	Item_10	Description of item 3	3.00E+000		5	1	3-Vector
18	Item_11	Description of item 4	4.00E+000		5	1	3-Vector
19	Item_12	Description of item 5	5.00E+000		5	1	3-Vector
20	Item_13	Description of item 6	6.00E+000		5	1	Scalar
21	Item_14	Description of item 7	7.00E+000		5	1	Scalar

Fig. 14.3-1: Generation of Application-Level Configuration Informationt

The next two subsections describe how the data pool meta-model was mapped to the Framework Configuration File. Note that a sample Framework Configuration File is available with the design data package of the CORDET Frameworks.

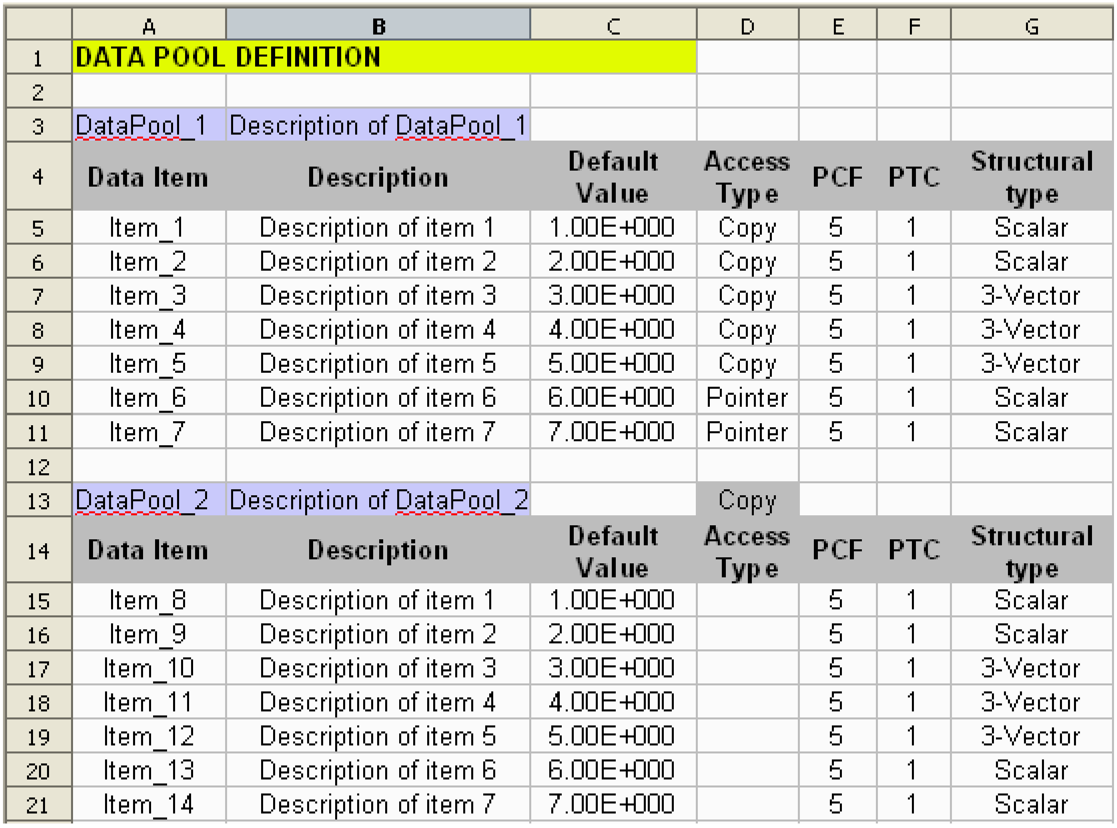
14.3.1 Data Pool Meta-Model Mapping

Spreadsheets have no explicit mechanisms for representing variability. Hence, the solution chosen for the data pool meta-model is to partially “flatten” the feature model and to map each terminal feature to a field in the spreadsheet.

Figure 14.3.1-1 shows a snapshot of the data pool part of the sample Framework Configuration File. The figure is nearly self-explanatory. The specific mapping rules for the data pool meta-model are:

- A data pool feature is mapped to set of consecutive records.
- A data item feature is mapped to a record
- The children sub-features of the data item feature are mapped to fields in the data item record
- The SyntacticalType field is split into two sub-fields representing the two sub-features of the SyntacticalType feature.
- The sub-features of the StructuralType and DataItemAccessMode features are mapped to enumerated values that are the only legal values of the homonymous fields in the data item records.
- The 'Size' features are mapped to enumerated values that are appended to the StructuralType and DataItemAccessMode values (in other words, they are just used as part of the names entered in the StructuralType and DataItemAccessMode fields).

- The 'current value' feature is not mapped to the spreadsheet since the value of this feature is not defined at design time.
- The global constraint on the access mode is mapped to a rule whereby the access mode can be defined either for the data pool as a whole (in the data pool record – as was done for the data pool 2 in figure 14.3.1-1) or for each data item (in the data item record – as was done for data pool 1 in figure 14.3.1-1).

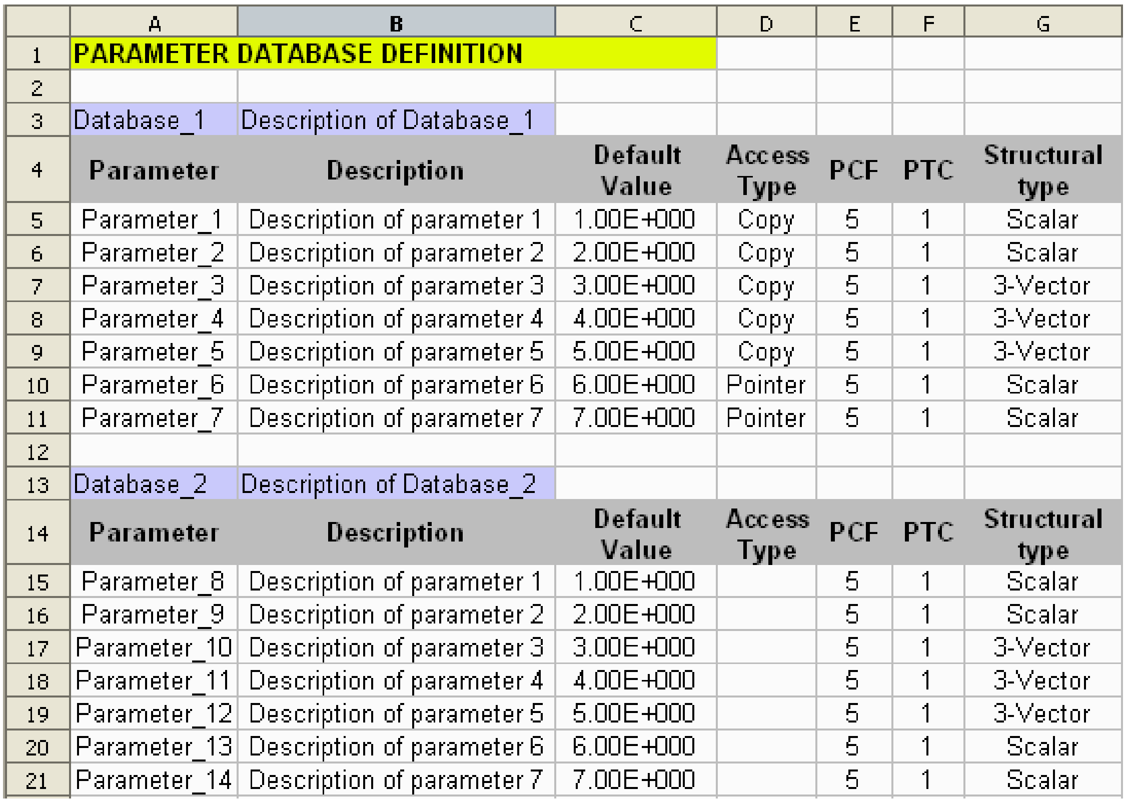


	A	B	C	D	E	F	G
1	DATA POOL DEFINITION						
2							
3	DataPool_1	Description of DataPool_1					
4	Data Item	Description	Default Value	Access Type	PCF	PTC	Structural type
5	Item_1	Description of item 1	1.00E+000	Copy	5	1	Scalar
6	Item_2	Description of item 2	2.00E+000	Copy	5	1	Scalar
7	Item_3	Description of item 3	3.00E+000	Copy	5	1	3-Vector
8	Item_4	Description of item 4	4.00E+000	Copy	5	1	3-Vector
9	Item_5	Description of item 5	5.00E+000	Copy	5	1	3-Vector
10	Item_6	Description of item 6	6.00E+000	Pointer	5	1	Scalar
11	Item_7	Description of item 7	7.00E+000	Pointer	5	1	Scalar
12							
13	DataPool_2	Description of DataPool_2		Copy			
14	Data Item	Description	Default Value	Access Type	PCF	PTC	Structural type
15	Item_8	Description of item 1	1.00E+000		5	1	Scalar
16	Item_9	Description of item 2	2.00E+000		5	1	Scalar
17	Item_10	Description of item 3	3.00E+000		5	1	3-Vector
18	Item_11	Description of item 4	4.00E+000		5	1	3-Vector
19	Item_12	Description of item 5	5.00E+000		5	1	3-Vector
20	Item_13	Description of item 6	6.00E+000		5	1	Scalar
21	Item_14	Description of item 7	7.00E+000		5	1	Scalar

Fig. 14.3.1-1: Snapshot of Data Pool Configuration Spreadsheet

14.3.2 Parameter Database Meta-Model Mapping

The mapping of the parameter database meta-model to the spreadsheet in the Framework Configuration File was done in manner analogous as for the data pool meta-model. The result is illustrated in figure 14.3.2-1 which shows a snapshot of the database part of the sample Framework Configuration File.



	A	B	C	D	E	F	G
1	PARAMETER DATABASE DEFINITION						
2							
3	Database_1	Description of Database_1					
4	Parameter	Description	Default Value	Access Type	PCF	PTC	Structural type
5	Parameter_1	Description of parameter 1	1.00E+000	Copy	5	1	Scalar
6	Parameter_2	Description of parameter 2	2.00E+000	Copy	5	1	Scalar
7	Parameter_3	Description of parameter 3	3.00E+000	Copy	5	1	3-Vector
8	Parameter_4	Description of parameter 4	4.00E+000	Copy	5	1	3-Vector
9	Parameter_5	Description of parameter 5	5.00E+000	Copy	5	1	3-Vector
10	Parameter_6	Description of parameter 6	6.00E+000	Pointer	5	1	Scalar
11	Parameter_7	Description of parameter 7	7.00E+000	Pointer	5	1	Scalar
12							
13	Database_2	Description of Database_2					
14	Parameter	Description	Default Value	Access Type	PCF	PTC	Structural type
15	Parameter_8	Description of parameter 1	1.00E+000		5	1	Scalar
16	Parameter_9	Description of parameter 2	2.00E+000		5	1	Scalar
17	Parameter_10	Description of parameter 3	3.00E+000		5	1	3-Vector
18	Parameter_11	Description of parameter 4	4.00E+000		5	1	3-Vector
19	Parameter_12	Description of parameter 5	5.00E+000		5	1	3-Vector
20	Parameter_13	Description of parameter 6	6.00E+000		5	1	Scalar
21	Parameter_14	Description of parameter 7	7.00E+000		5	1	Scalar

Fig. 14.3.2-1: Snapshot of Parameter Database Configuration Spreadsheet

14.3.3 Other Configuration Information

As indicated above, the Framework Configuration File is intended to hold configuration information about other parts of the framework in addition to the data pool and parameter database. In all cases, the principle is the same: part of the feature model that defines the framework is mapped to a spreadsheet. This spreadsheet is intended for the application designer as a means to define the configuration of his application in terms of the features offered by the framework.

The sample Framework Configuration File contains three additional spreadsheets in addition to those for the data pool and parameter database:

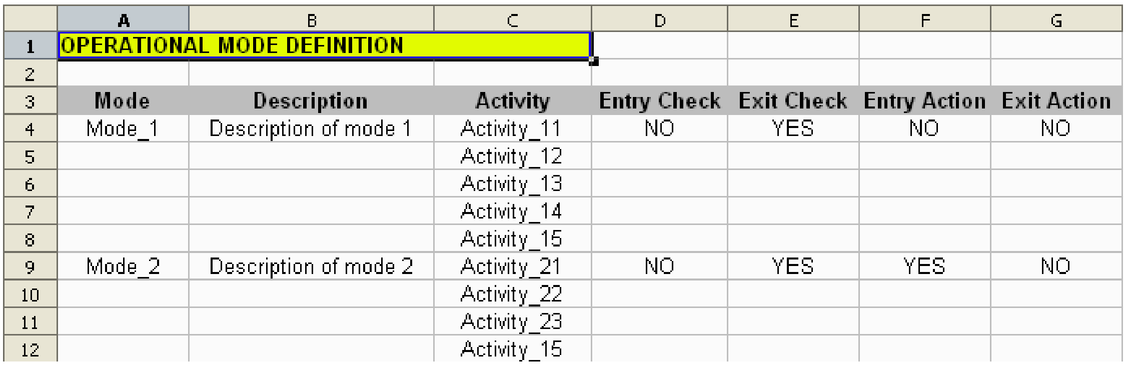
- The *OperationalMode* spreadsheet defines the configuration of the operational modes in an application.
- The *Activity* spreadsheet defines the configuration of the activities in an application.
- The *HealthChecks* spreadsheet defines the configuration of the health checks in an application.

Note that whereas the spreadsheets for the data pool and database are intended to capture all the information required to define the data pool and database of an application, in the cases above, instead, the spreadsheet are intended to only indicate which of the variable features offered by the framework are effectively used at application level.

The specific mapping rules for the parameter database meta-model are:

- A parameter database feature is mapped to set of consecutive records.
- A parameter feature is mapped to a record

- The children sub-features of the parameter feature are mapped to fields in the parameter record
- The SyntacticalType field is split into two sub-fields representing the two sub-features of the SyntacticalType feature.
- The sub-features of the StructuralType and ParameterAccessMode features are mapped to enumerated values that are the only legal values of the homonymous fields in the parameter records.
- The 'Size' features are mapped to enumerated values that are appended to the StructuralType and ParameterAccessMode values (in other words, they are just used as part of the names entered in the StructuralType and ParameterAccessMode fields).
- The global constraint on the access mode is mapped to a rule whereby the access mode can be defined either for the database as a whole (in the parameter database record – as was done for the database 2 in figure 14.3.3-1) or for each parameter (in the parameter – as was done for database 1 in figure 14.3.3-1).

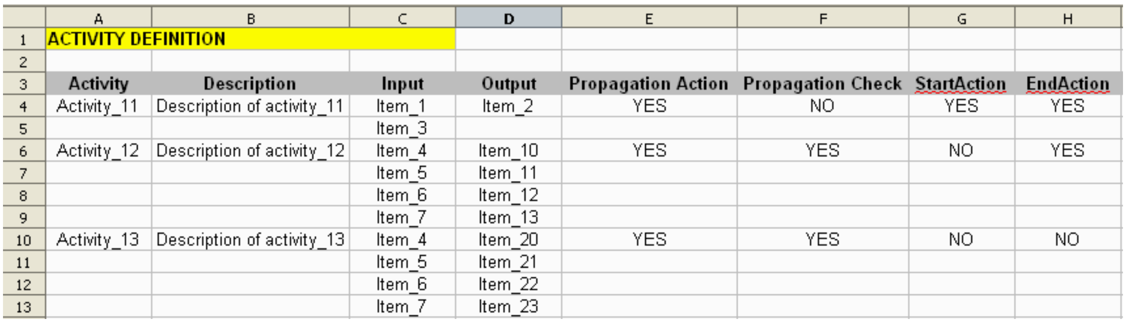


	A	B	C	D	E	F	G
1	OPERATIONAL MODE DEFINITION						
2							
3	Mode	Description	Activity	Entry Check	Exit Check	Entry Action	Exit Action
4	Mode_1	Description of mode 1	Activity_11	NO	YES	NO	NO
5			Activity_12				
6			Activity_13				
7			Activity_14				
8			Activity_15				
9	Mode_2	Description of mode 2	Activity_21	NO	YES	YES	NO
10			Activity_22				
11			Activity_23				
12			Activity_15				

Fig. 14.3.3-1: Snapshot of Operational Model Configuration Spreadsheet

As an example, figure 14.3.3-1 shows the configuration spreadsheet for the operational modes. This spreadsheet shows at a glance which operational modes have been defined for a certain application, which activities are attached to each operational mode, and which checks and actions are implemented for each operational mode. The spreadsheet could be further expanded to give a specification (or perhaps a pointer to a specification) of the checks and actions.

As a final example, figure 14.3.3-2 shows the configuration spreadsheet for the activities. Note that the information in the various spreadsheet can be cross-linked to ensure consistency (thus, for instance, the activity entries in figure 14.3.3-1 are pointers to the activity definitions in figure 14.3.3-2).



	A	B	C	D	E	F	G	H
1	ACTIVITY DEFINITION							
2								
3	Activity	Description	Input	Output	Propagation Action	Propagation Check	StartAction	EndAction
4	Activity_11	Description of activity_11	Item_1	Item_2	YES	NO	YES	YES
5			Item_3					
6	Activity_12	Description of activity_12	Item_4	Item_10	YES	YES	NO	YES
7			Item_5	Item_11				
8			Item_6	Item_12				
9			Item_7	Item_13				
10	Activity_13	Description of activity_13	Item_4	Item_20	YES	YES	NO	NO
11			Item_5	Item_21				
12			Item_6	Item_22				
13			Item_7	Item_23				

Fig. 14.3.3-2: Snapshot of Activity Configuration Spreadsheet

14.4 Code Generator for Data Pool and Parameter Database

The Control Framework proposes a design-level model for the data pool and parameter database concepts that is based on a meta-model. This implies that the data pool and parameter database design models to be used in a concrete application should be obtained by instantiating this meta-model.

The instantiation of the meta-model can be done either manually or automatically using a code generator. The issue naturally arises of whether the decision as to which technique to use – manual or automatic generation – should be done at framework or at application level. Also, in the case where a code generator is to be used, the related issue arises of whether this should be defined at framework or at application level.

The OBS Framework, one of the predecessors of the Control Framework (see reference RD-18), took the view that the data pool and parameter database ought to be built automatically from their design-level models and that the code generator ought to be defined at framework level. There are now two facts that militate against the same choice for the CORDET Control Framework:

1. The experience from the OBS Framework Project is that framework-level code generator are very complex to build (because they must encompass all possible structural choices for the target data pool or parameter database)
2. Since the time when the OBS Framework was defined (2003-4), the use of code generators has become more widespread and more widely accepted by programmers.

For the CORDET Project, the decision has therefore been taken to treat the generation of data pool and parameter databases as an application-level issue. The data pools and parameter databases, in other words, are treated in the same way as the non-functional containers: namely as structures that are generated at application-instantiation time from a meta-model defined at framework level.