305-CD-047-001

EOSDIS Core System Project

Flight Operations Segment (FOS) Analysis Design Specification for the ECS Project

October 1995

Hughes Information Technology Corporation
Upper Marlboro, MD

Flight Operation Segment (FOS) Analysis Design Specification for the ECS Project

October 1995

Prepared Under Contract NAS5-60000
CDRL Item #046

APPROVED BY

Cal E. Moore, Jr. /s/ 9/25/95
Cal Moore, FOS CCB Chairman Date
EOSDIS Core System Project

Hughes Information Technology Corporation
Upper Marlboro, Maryland

305-CD-047-001

This page intentionally left blank.

Preface

This document, one of nineteen, comprises the detailed design specification of the FOS subsystems for Releases A and B of the ECS project. This includes the FOS design to support the AM-1 launch.

The FOS subsystem design specification documents for Releases A and B of the ECS project include:

305-CD-040 FOS Design Specification (Segment Level Design)
305-CD-041 Planning and Scheduling Design Specification
305-CD-042 Command Management Design Specification
305-CD-043 Resource Management Design Specification
305-CD-044 Telemetry Design Specification
305-CD-045 Command Design Specification
305-CD-046 Real-Time Contact Management Design Specification
305-CD-047 Analysis Design Specification
305-CD-048 User Interface Design Specification
305-CD-049 Data Management Design Specification
305-CD-050 Planning and Scheduling Program Design Language (PDL)
305-CD-051 Command Management PDL
305-CD-052 Resource Management PDL
305-CD-053 Telemetry PDL
305-CD-054 Real-Time Contact Management PDL
305-CD-055 Analysis PDL
305-CD-056 User Interface PDL
305-CD-057 Data Management PDL
305-CD-058 Command PDL

Object models presented in this document have been exported directly from CASE tools and in some cases contain too much detail to be easily readable within hard copy page constraints. The reader is encouraged to view these drawings on line using the Portable Document Format (PDF) electronic copy available via the ECS Data Handling System (EDHS) at URL http://edhs1.gsfc.nasa.gov.

This document is a contract deliverable with an approval code 2. As such, it does not require formal Government approval, however, the Government reserves the right to request changes within 45 days of the initial submittal. Once approved, contractor changes to this document are handled in accordance with Class I and Class II change control requirements described in the EOS Configuration Management Plan, and changes to this document shall be made by document change notice (DCN) or by complete revision.

Any questions should be addressed to:

Data Management Office
The ECS Project Office
Hughes Information Technology Corporation
1616 McCormick Drive
Upper Marlboro, Maryland 20774-5372

Abstract

The FOS Design Specification consists of a set of 19 documents that define the FOS detailed design. The first document, the FOS Segment Level Design, provides an overview of the FOS segment design, the architecture, and analyses and trades. The next nine documents provide the detailed design for each of the nine FOS subsystems. The last nine documents provide the PDL for the nine FOS subsystems. It also allocates the level 4 FOS requirements to the subsystem design.

Keywords: FOS, design, specification, analysis, IST, EOC

This page intentionally left blank.

Change Information Page

List of Effective Pages	
Page Number	**Issue**
Title	Original
iii through xii	Original
1 -1 and 1-2	Original
2-1 through 2-4	Original
3-1 through 3-126	Original
AB-1 through AB-8	Original
GL-1 through GL-8	Original

Document History			
Document Number	**Status/Issue**	**Publication Date**	**CCR Number**
305-CD-047-001	Original	October 1995	95-0672

This page intentionally left blank.

Contents

Preface

Abstract

1. Introduction

1.1 Identification 1-1
1.2 Scope 1-1
1.3 Purpose 1-1
1.4 Status and Schedule 1-1
1.5 Document Organization 1-1

2. Related Documentation

2.1 Parent Document 2-1
2.2 Applicable Documents 2-1
2.3 Information Documents 2-2
2.3.1 Information Document Referenced 2-2

3. Analysis Subsystem

3.1 Analysis Context 3-1
3.2 Offline Analysis Processing 3-4
3.2.1 Offline Analysis Processing Context 3-4
3.2.2 Offline Analysis Processing Interfaces 3-6
3.2.3 Offline Analysis Object Model 3-6
3.2.4 Offline Analysis Dynamic Model 3-24
3.2.5 Offline Analysis Data Dictionary 3-34
3.3 Analysis Request Manager 3-66
3.3.1 Analysis Request Manager Context 3-66
3.3.2 Analysis Request Manager Interface 3-68
3.3.3 Analysis Request Manager Object Model 3-68
3.3.4 Analysis Request Manager Dynamic Model 3-70
3.3.5 Analysis Requests Processing Data Dictionary 3-72

3.4 Spacecraft Clock Correlation Analysis 3-78
3.4.1 Clock Correlation Analysis Context 3-80
3.4.2 Clock Correlation Analysis Interfaces 3-82
3.4.3 Clock Correlation Analysis Object Model 3-82
3.4.4 Clock Correlation Analysis Dynamic Model 3-85
3.4.5 Clock Correlation Data Dictionary 3-90
3.5 Solid State Recorder Analysis Processing 3-96
3.5.1 Solid State Recorder Analysis Processing Context 3-96
3.5.2 Solid State Recorder Analysis Processing Interfaces 3-98
3.5.3 SSR Analysis Processing Object Model 3-98
3.5.4 Solid State Recorder Analysis Processing Dynamic Model 3-100
3.5.5 Solid State Recorder Analysis Processing Data Dictionary 3-111
3.6 User Algorithms 3-114
3.6.1 User Algorithms Context 3-114
3.6.2 User Algorithms Interfaces 3-116
3.6.3 User Algorithms Object Model 3-116
3.6.4 User Algorithms Dynamic Model 3-119
3.6.5 User Algorithms Data Dictionary 3-121

Figures

3.1-1. Analysis Context 3-3
3.2-1. Offline Analysis Context 3-5
3.2-2. Offline Analysis Processing with External Interfaces 3-8
3.2-3. Offline Analysis Processing with Internal Components 3-9
3.2-4. Offline Analysis Products 3-10
3.2-5. Offline Analysis Parameter Groups 3-12
3.2-6. Offline Analysis Parameters 3-13
3.2-7. Offline Analysis - Analysis Parameters 3-14
3.2-8. Offline Analysis Limits and Discrete State Parameters 3-16
3.2-9. Offline Analysis Statistics Parameters 3-17
3.2-10. Offline Analysis Telemetry Request Product 3-18
3.2-11. Offline Analysis System Generated Statistics 3-20
3.2-12. Offline Analysis System Generated Limits Statistics 3-21
3.2-13. Offline Analysis System Generated Discrete Statistics 3-22
3.2-14. Analysis Request Object Model 3-23
3.2-15. Basic Dataset Generation Event Trace 3-25
3.2-16. System Generated Analog Statistics Event Trace 3-27
3.2-17. Daily Stats from Orbital Stats Event Trace 3-29

3.2-18. System Generated Limits Statistics Event Trace 3-31
3.2-19. System Generated Discrete Statistics Event Trace 3-33
3.3-1. Analysis Request Manager Context 3-67
3.3-2. Analysis Request Manager Object Model 3-69
3.3-3. Analysis Request Manager Event Trace 3-71
3.4-1. Clock Correlation Analysis High Level Data Flow 3-79
3.4-2. Clock Correlation Analysis Context Diagram 3-81
3.4-3. Clock Correlation Analysis Object Model 3-84
3.4-4. USCCS Clock Correlation Processing Event Trace 3-87
3.4-5. RDD Clock Correlation Processing Event Trace 3-89
3.5-1. Solid State Recorder Analysis Processing Context 3-97
3.5-2. Solid State Recorder Analysis Processing Object Model 3-99
3.5-3. SSR Playback Nominal Event Trace 3-102
3.5-4. SSR Replay Loss of Data Replay NOT in Current Contact Event Trace 3-105
3.5-5. SSR Replay Loss of Data Replay During Current Contact Event Trace 3-108
3.5-6. Solid State Recorder Analysis Processing State Diagram 3-110
3.6-1. User Algorithms Context 3-115
3.6-2. User Algorithms Object Model 3-117
3.6-3. User Algorithms Object Model 3-118
3.6-4. User Algorithms Event Trace 3-120

Tables

3.2.2. Offline Analysis Processing Interfaces 3-6
3.3.2. Analysis Request Manager Interface 3-68
3.4.2. Clock Correlation Analysis Interfaces 3-82
3.5.2. Solid State Recorder Analysis Processing Interfaces 3-98

Abbreviations and Acronyms

Glossary

This page intentionally left blank.

1. Introduction

1.1 Identification

The contents of this document defines the design specification for the Flight Operations Segment (FOS). Thus, this document addresses the Data Item Description (DID) for CDRL Item 046 305/DV2 under Contract NAS5-60000.

1.2 Scope

The Flight Operations Segment (FOS) Design Specification defines the detailed design of the FOS. It allocates the Level 4 FOS requirements to the subsystem design. It also defines the FOS architectural design. In particular, this document addresses the Data Item Description (DID) for CDRL # 046, the Segment Design Specification.

This document reflects the August 23, 1995 Technical Baseline maintained by the contractor configuration control board in accordance with ECS Technical Direction No. 11, dated December 6, 1994. It covers releases A and B for FOS. This corresponds to the design to support the AM-1 launch.

1.3 Purpose

The FOS Design Specification consists of a set of 19 documents that define the FOS detailed design. The first document, the FOS Segment Level Design, provides an overview of the FOS segment design, the architecture, and analyses and trades. The next nine documents provide the detailed design for each of the nine FOS subsystems. The last nine documents provide the PDL for the nine FOS subsystems.

1.4 Status and Schedule

This submittal of DID 305/DV2 incorporates the FOS detailed design performed during the Critical Design Review (CDR) time frame. This document is under the ECS Project configuration control.

1.5 Document Organization

305-CD-040 contains the overview, the FOS segment models, the FOS architecture, and FOS analyses and trades performed during the design phase.

305-CD-041 contains the detailed design for Planning and Scheduling Design Specification.

305-CD-042 contains the detailed design for Command Management Design Specification.

305-CD-043 contains the detailed design for Resource Management Design Specification.

305-CD-044 contains the detailed design for Telemetry Design Specification.

305-CD-045 contains the detailed design for Command Design Specification.

305-CD-046 contains the detailed design for Real-Time Contact Management Design Specification.

305-CD-047 contains the detailed design for Analysis Design Specification.

305-CD-048 contains the detailed design for User Interface Design Specification.

305-CD-049 contains the detailed design for Data Management Design Specification.

305-CD-050 contains Planning and Scheduling PDL.

305-CD-051 contains Command Management PDL.

305-CD-052 contains Resource Management PDL.

305-CD-053 contains the Telemetry PDL.

305-CD-054 contains the Real-Time Contact Management PDL.

305-CD-055 contains the Analysis PDL.

305-CD-056 contains the User Interface PDL.

305-CD-057 contains the Data Management PDL.

305-CD-058 contains the Command PDL.

Appendix A of the first document contains the traceability between Level 4 Requirements and the design. The traceability maps the Level 4 requirements to the objects included in the subsystem object models.

Glossary contains the key terms that are included within this design specification.

Abbreviations and acronyms contains an alphabetized list of the definitions for abbreviations and acronyms used within this design specification.

2. Related Documentation

2.1 Parent Document

The parent documents are the documents from which this FOS Design Specification's scope and content are derived.

194-207-SE1-001	System Design Specification for the ECS Project
304-CD-001-002	Flight Operations Segment (FOS) Requirements Specification for the ECS Project, Volume 1: General Requirements
304-CD-004-002	Flight Operations Segment (FOS) Requirements Specification for the ECS Project, Volume 2: AM-1 Mission Specific

2.2 Applicable Documents

The following documents are referenced within this FOS Design Specification or are directly applicable, or contain policies or other directive matters that are binding upon the content of this volume.

194-219-SE1-020	Interface Requirements Document Between EOSDIS Core System (ECS) and NASA Institutional Support Systems
209-CD-002-002	Interface Control Document Between EOSDIS Core System (ECS) and ASTER Ground Data System, Preliminary
209-CD-003-002	Interface Control Document Between EOSDIS Core System (ECS) and the EOS-AM Project for AM-1 SpaceCraft Analysis Software, Preliminary
209-CD-004-002	Data Format Control Document for the Earth Observing System (EOS) AM-1 Project Data Base, Preliminary
209-CD-025-001	ICD Between ECS and AM1 Project Spacecraft Software Development and Validation Facilities (SDVF)
311-CD-001-003	Flight Operations Segment (FOS) Database Design and Database Schema for the ECS Project
502-ICD-JPL/GSFC	Goddard Space Flight Center/MO&DSD, Interface Control Document Between the Jet Propulsion Laboratory and the Goddard Space Flight Center for GSFC Missions Using the Deep Space Network
530-ICD-NCCDS/MOC	Goddard Space Flight Center/MO&DSD, Interface Control Document Between the Goddard Space Flight Center Mission Operations Centers and the Network Control Center Data System
530-ICD-NCCDS/POCC	Goddard Space Flight Center/MO&DSD, Interface Control Document Between the Goddard Space Flight Center Payload Operations Control Centers and the Network Control Center Data System

530-DFCD-NCCDS/POCC	Goddard Space Flight Center/MO&DSD, Data Format control Document Between the Goddard Space Flight Center Payload Operations Control Centers and the Network Control Center Data System
540-041	Interface Control Document (ICD) Between the Earth Observing System (EOS) Communications (Ecom) and the EOS Operations Center (EOC), Review
560-EDOS-0230.0001	Goddard Space Flight Center/MO&DSD, Earth Observing System (EOS) Data and Operations System (EDOS) Data Format Requirements Document (DFRD)
ICD-106	Martin Marietta Corporation, Interface Control Document (ICD) Data Format Control Book for EOS-AM Spacecraft
none	Goddard Space Flight Center, Earth Observing System (EOS) AM-1 Flight Dynamics Facility (FDF) / EOS Operations Center (EOC) Interface Control Document

2.3 Information Documents

2.3.1 Information Document Referenced

The following documents are referenced herein and, amplify or clarify the information presented in this document. These documents are not binding on the content of this FOS Design Specification.

194-201-SE1-001	Systems Engineering Plan for the ECS Project
194-202-SE1-001	Standards and Procedures for the ECS Project
193-208-SE1-001	Methodology for Definition of External Interfaces for the ECS Project
308-CD-001-004	Software Development Plan for the ECS Project
194-501-PA1-001	Performance Assurance Implementation Plan for the ECS Project
194-502-PA1-001	Contractor's Practices & Procedures Referenced in the PAIP for the ECS Project
604-CD-001-004	Operations Concept for the ECS Project: Part 1-- ECS Overview, 6/95
604-CD-002-001	Operations Concept for the ECS project: Part 2B -- ECS Release B, Annotated Outline, 3/95
604-CD-003-001	ECS Operations Concept for the ECS Project: Part 2A -- ECS Release A, Final, 7/95
194-WP-912-001	EOC/ICC Trade Study Report for the ECS Project, Working Paper
194-WP-913-003	User Environment Definition for the ECS Project, Working Paper
194-WP-920-001	An Evaluation of OASIS-CC for Use in the FOS, Working Paper
194-TP-285-001	ECS Glossary of Terms
222-TP-003-006	Release Plan Content Description

none	Hughes Information Technology Company, Technical Proposal for the EOSDIS Core System (ECS), Best and Final Offer
560-EDOS-0211.0001	Goddard Space Flight Center, Interface Requirements Document (IRD) Between the Earth Observing System (EOS) Data and Operations System (EDOS), and the EOS Ground System (EGS) Elements, Preliminary
NHB 2410.9A	NASA Hand Book: Security, Logistics and Industry Relations Division, NASA Security Office: Automated Information Security Handbook

This page intentionally left blank.

3. Analysis Subsystem

The Analysis Subsystem is responsible for the analysis required for operations of the U.S. spacecraft, its subsystems, and its instrument payload. This analysis includes trend analysis, performance analysis, configuration monitoring, resource management, and fault management. The analysis can be performed using real time or operations history data. The analysis results are used as input to overall mission monitoring.

The Analysis Subsystem will provide behavior analysis for each spacecraft subsystem to maintain an optimal and well understood level of performance, including command and data handling, electrical power, thermal, propulsion, solid state recorder, and guidance navigation and control. The instrument engineering team will track changes in instrument behavior. Anomalies uncovered as a result of instrument behavior analysis will be reported to the EOC, along with procedures for dealing with the anomalies.

3.1 Analysis Context

The interfaces between the FOS Analysis Subsystem (FAS) and other FOS subsystems are illustrated by the context diagram in Figure 3.1-1. Descriptions of each interface are summarized as follows:

FOS User Interface Subsystem (FUI) - Off-line Analysis can be initiated via a request generated by FUI. Requests can be made as needed, at a specified time interval (e.g., every day, every month, once per orbit, etc), or in response to selected events. FAS provides the status of the request to FUI, including a percentage of the processing completed. After the processing is completed, the data is formatted into a dataset or report and sent to FUI to be displayed. Additionally, if FAS detects a problem during the SSR playback, it sends recommended recovery procedures to FUI to be displayed to the operator. After each contact FAS provides FUI with a SSR status report.

FOS Telemetry Subsystem (TLM) - Real-time and historical telemetry values are made available to FAS via TLM. TLM selectively decommutates the parameters requested by FAS and provides the raw and EU converted values along with associated data quality flags.

FOS Data Management Subsystem (DMS) - Since a request for historical data could span multiple versions of the database, DMS provides FAS with information about the database versions and the time periods for which they are active. FAS uses this information to determine which database TLM should use to process the telemetry. FAS also receives FDF data and archived statistics from DMS. Events generated by FAS are sent to DMS for processing. Statistics that are automatically generated are stored by DMS to be used in future analysis requests. DMS notifies FAS when the back orbit data and FDG data has been archived so that statistics can be generated.

FOS Real-time Contact Management Subsystem (RCM) - NCC and EDOS data is made available to FAS via RCM.

FOS Resource Management Subsystem (RMS) - FAS requests telemetry data via RMS. During a contact, FAS can notify RMS that it needs to receive real-time telemetry. In order to get historical telemetry, FAS tells RMS the start and stop times for the data and RMS requests it from FDM and configures TLM to process it.

Planning and Scheduling (PAS) - Before each contact FAS requests predict data from PAS. After each contact FAS provides Planning and Scheduling with a SSR status report.

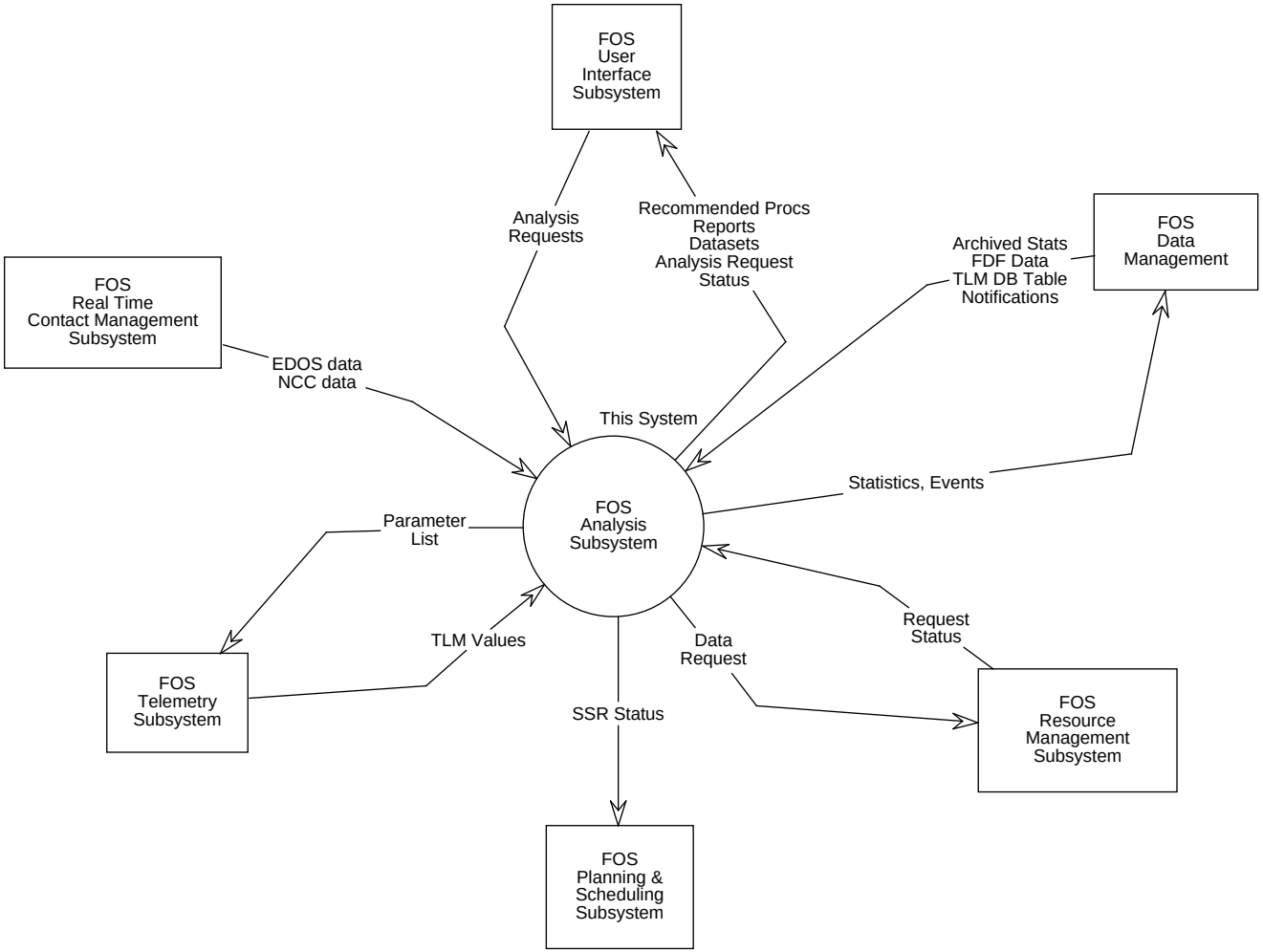

Figure 3.1-1. Analysis Context

3.2 Offline Analysis Processing

Offline Analysis Processing is the actual number crunching of spacecraft telemetry. It produces datasets for use by FUI, carry-out data for use by FOS and non-FOS engineers, and statistics for long term trending of spacecraft conditions.

3.2.1 Offline Analysis Processing Context

The Offline Analysis process interacts with the Analysis Request Manager process, DataServer process (DMS), DECOM process, ParameterServer object to process the Analysis request and generate the telemetry datasets. There will be one instance of FaAcOfflineAnaController per request.

Refer to Offline Analysis Context Diagram Figure 3.2-1;

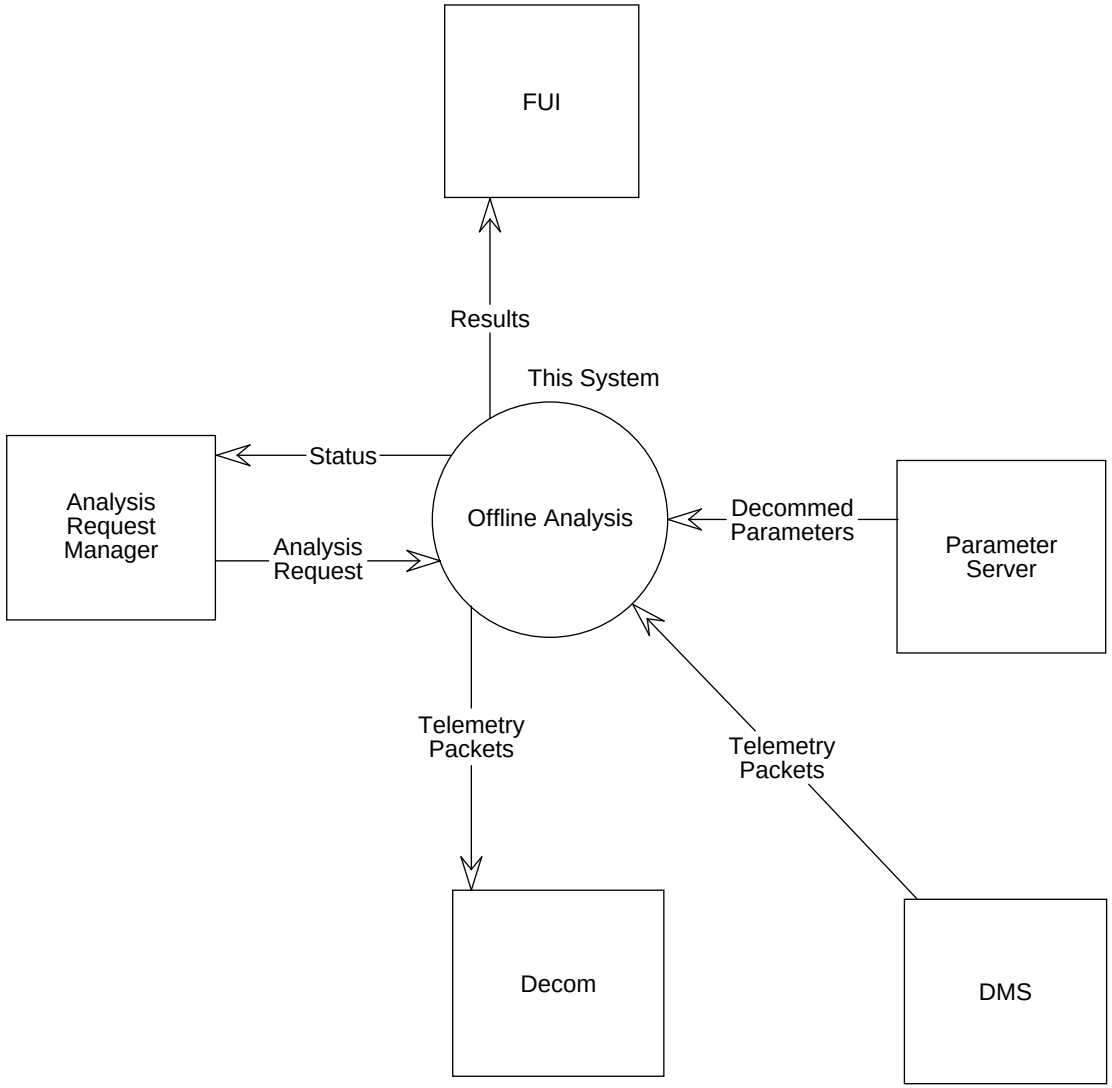

Figure 3.2-1. Offline Analysis Context

3.2.2 Offline Analysis Processing Interfaces

***Table* 3.2.2. Offline Analysis Processing Interfaces**

Interface Service	Interface Class	Interface Class Description	Service Provider	Service User	Frequency
Receive EDUs	FtTlEdu	Receives EDUs from Offline Analysis process	TLM	ANA	As many as number of EDUs
Receive EDU Request	F0ReEDURequest	Receives EDU request from Offline Analysis process	DMS	ANA	As many as number of EDUs
Send Analysis Request	FaRpReqMgToAna Proxy	Submits Analysis request from the Request Manager to the Offline Analysis process	ANA	ANA	Once per request

3.2.3 Offline Analysis Object Model

There are several object models associated with the Offline Analysis process:

- Offline Analysis processing with reference to external interfaces
- Offline Analysis processing with internal components
- Offline Analysis Products
- Offline Analysis Parameter Groups
- Offline Analysis Parameters
- Offline Analysis - Analysis Parameter
- Offline Analysis Limits and Discrete State Parameters
- Offline Analysis Statistics Parameters
- Offline Analysis System Generated Statistics
- Offline Analysis System Generated Limits Statistics
- Offline Analysis System Generated Discrete State Statistics
- Analysis Request

3.2.3.1 Offline Analysis Processing with Reference to External Interfaces

This model shows the relationship between the external interfaces and Offline Analysis Controller Object. The FaAcOfflineAnaController object receives an Analysis request from FaRmAnalysisRequestManager object. Based on the Analysis Request, it instantiates all the parameters specified in the request. It also instantiates a ParameterMonitor object. It requests raw data from DMS by using FoReEDU Request and receives it from DMS. It sends an EDU to the DECOM process via the FtTlEDU interface object. The DECOM process decommutates the raw data and sends processed parameters to the FaAcOfflineAnaController via the ParameterMonitor

object. The FaAcOfflineAnaController then notifies FaPrAnalysisproducts about the newly decommutated parameters. When all the data is processed by FaPrAnalysisProduct objects, the FaAcOfflineAnaController notifies Analysis Request Manager about the completion and then deletes the FaAcOfflineAnaController object. The Offline Analysis Controller process, then terminates.

-Refer to Offline Analysis processing with reference to external interfaces - Figure 3.2-2;

3.2.3.2 Offline Analysis Processing with Internal Components

This model shows the relation ship between the FaAcOfflineAnaController with that of internal components of the Analysis process. Upon receiving the Analysis request, the FaAcOfflineAnaController instantiates FaPrAnalysisProduct object which in turn instantiates FaPaParameterGroup objects based on the requested parameters in the Analysis request. The FaPaParameterGroup instantiates FaPaAnalysisParameter objects corresponding to each parameter in the FaPaParameterGroup objects. As the decommutated parameters are notified to the FaAcOfflineAnaController, it notifies FaPrAnalysisProduct to update. The FaPrAnalysisProduct inturn notifies FaPaParameterGroup to update. The FaPaParameterGroup uses the newly arrived parameters for the generation of products. The process of receiving new parameters and product generation continues till the request time span completes. The FaAcOfflineAnaController notifies Analysis Request Manager about the completion of request and then the Offline Analysis process terminates.

-Refer to Offline Analysis processing with internal components - Figure 3.2-3;

3.2.3.3 Offline Analysis Products

This model shows the different products generated by the offline analysis process. The FaPrAnalysisProduct is a base class for the analysis products. The FaPrDataset is a base class for the different datasets produced by the offline analysis process. The objects derived from the base class are: FaPrSystemStatsDataset, FaPrLimitsDataset, FaPrDiscreteStateDataset, FaPrOrbitSystemsStatsDataset, FaPrDailySystemStatsDataset, FaPrMonthlySystemsStatsDataset etc. The FaPrSystemStatsDataset generates the system statistics like daily, monthly, per orbit, per orbit day, per orbit night etc. The FaPrDatasetReader reads different datasets to generate plots or spread sheets or a report.

-Refer to Offline Analysis Products - Figure 3.2-4;

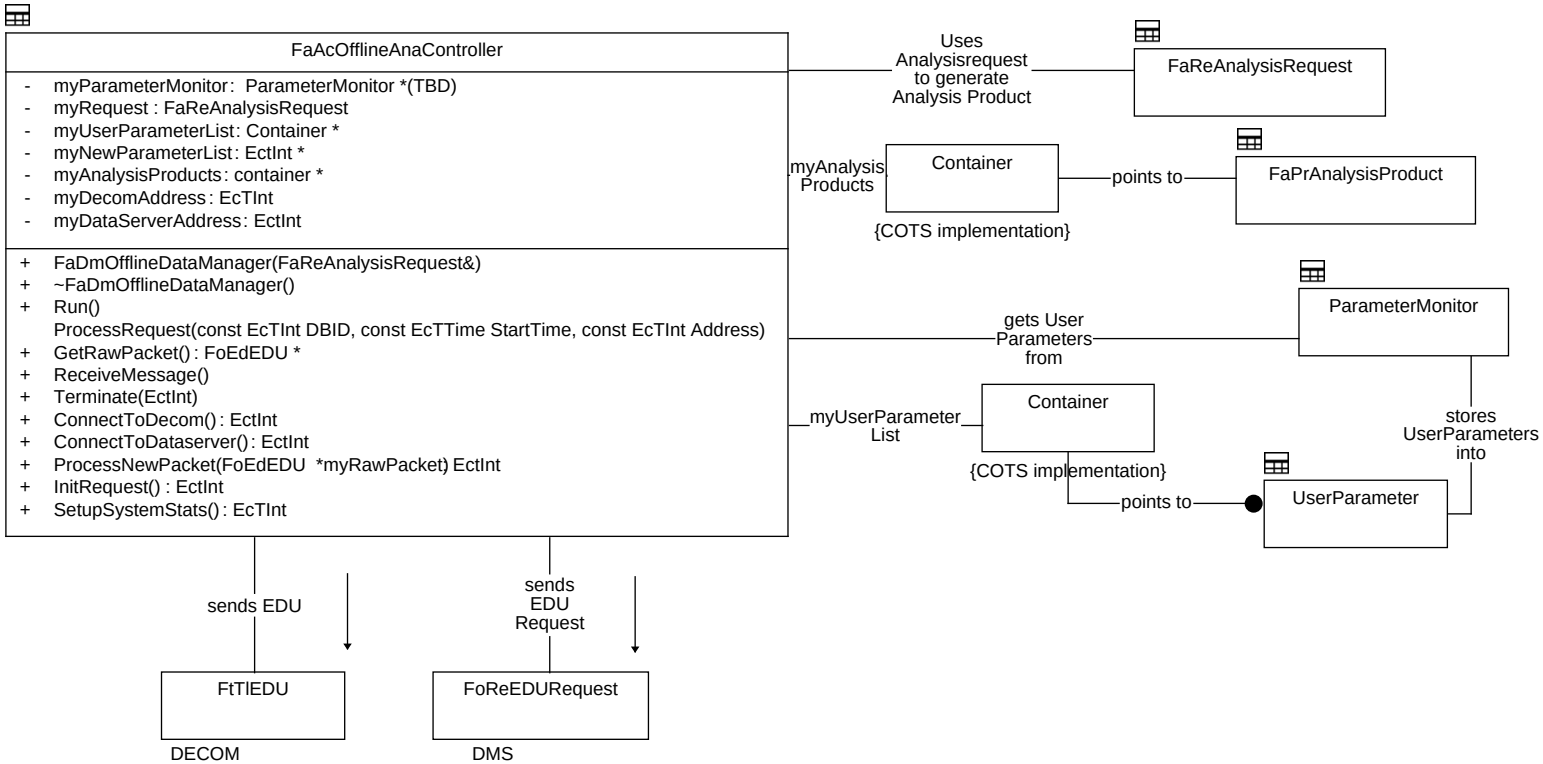

Figure 3.2-2. Offline Analysis Processing with External Interfaces

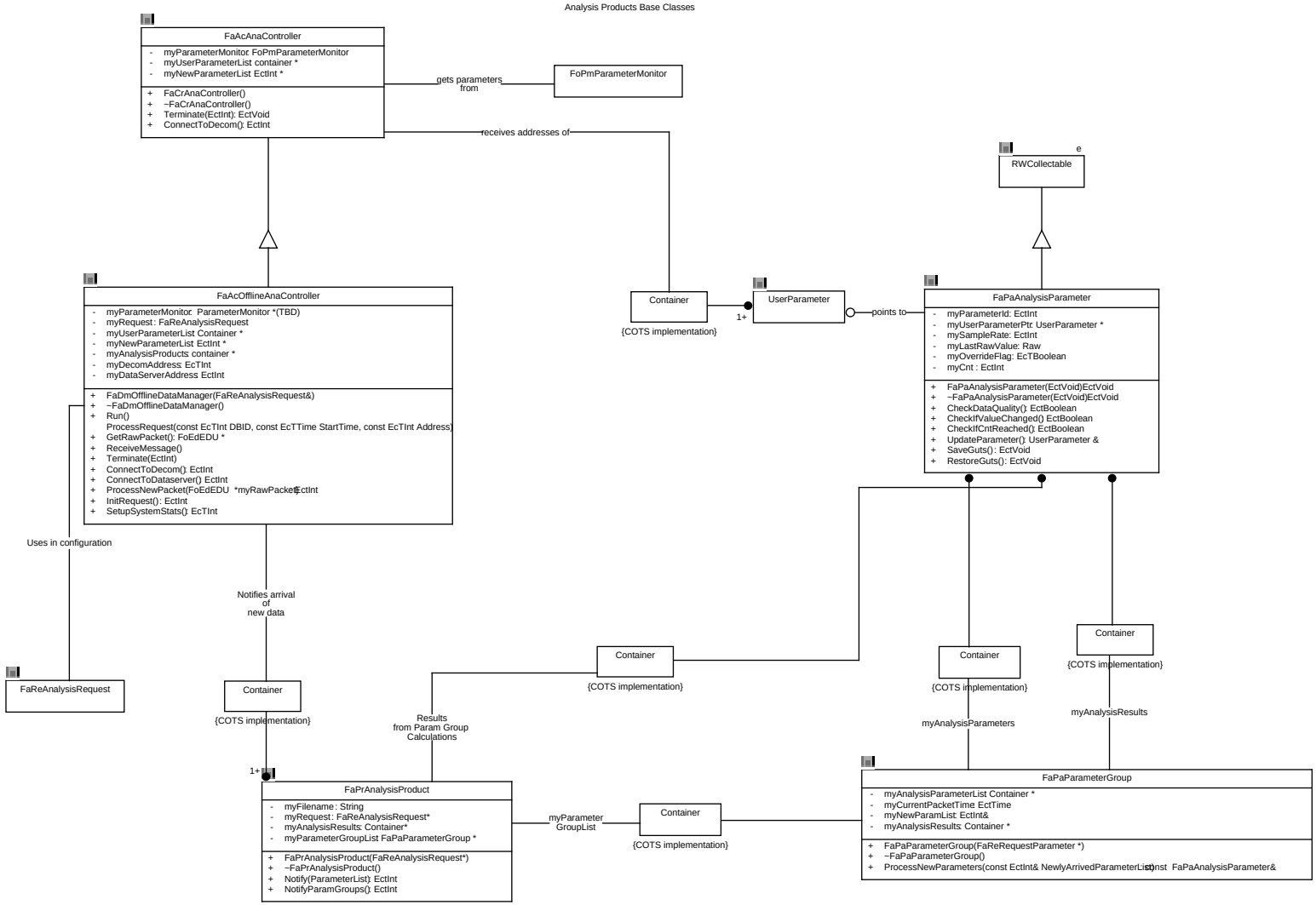

Figure 3.2-3. Offline Analysis Processing with Internal Components

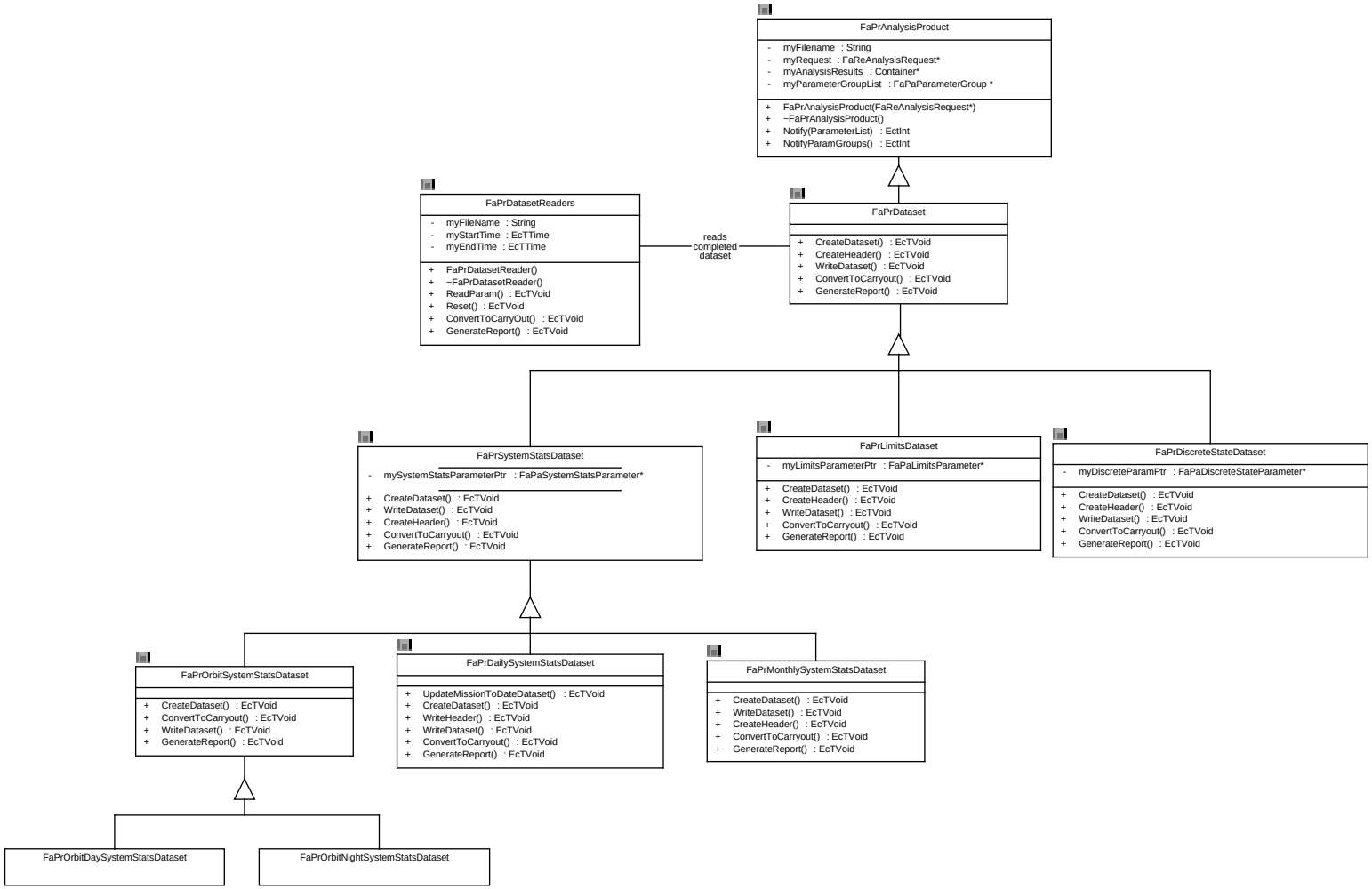

Figure 3.2-4. Offline Analysis Products

3.2.3.4 Offline Analysis Parameter Groups

This model shows different parameter group classes which are used for controlling associated products and parameters. The FaPaParameterGroup is base class for all other parameter group classes. A parameter group is a collection of parameters which perform some function and return zero or more parameters as a result. The FaPaSubsystemParameterGroup class is a collection of telemetry parameters and algorithms used to generate data for a subsystem report. The FaPaSystemStatsParamGroup class is a base class for the system generated statistics. The FaPaUserStatsParamGroup is a group of user stats parameters. Each user stats parameter has an user specified interval, from 1 sec to 24 hours when the statistics are generated. The FaPaAnalogStatsParamGroup class is a collection of analog parameters which are used for the generation of orbital statistics. The FaPaDiscreteStateParamGroup is used for the generation of the system generated statistics for the discrete parameters. For each discrete parameter, the number of state changes and total time spent in each state is kept for the day, month, and MTD. The FaPaLimitsParamGroup is used for the generation of limits statistics for all the parameters. The FaPaDLOParanGroup sorts the parameters in time order, so they can be printed in a Downlink ordered Report.

-Refer to Offline Analysis Parameter Groups - Figure 3.2-5;

3.2.3.5 Offline Analysis Parameters

This model shows the different parameter classes used in the generation of Analysis products. The FaPaAnalysisParameter is the base class for all other parameter classes. This class checks the data quality and sample rate before passing the parameter address to the parameter group. The FaPaStatsParameter class is the base class for the FaPaUserStatsParameter and FaPaAnalogStatsParameter. The FaPaUserStatsParameter accumulates statistical values for the user defined interval (from 1 sec to 24 hours). The FaPaAnalogStatsParameter generates system generated statistics for Analog parameters. The FaPaLimitsParameter detects the limit violation and computes the limits violation duration for each telemetry parameter. The FaPaDiscreteStateParameter detects the state change and computes the state change duration. It also generates the daily statistics of state changes for each discrete parameter. The limit violations and discrete state changes are computed when the system generated statistics request is executed.
-Refer to Offline Analysis Parameters - Figure 3.2-6;

3.2.3.6 Offline Analysis - Analysis Parameter

This model presents the Analysis parameter class which is the base class for all other parameter classes. It has a pointer to a user parameter class which provides the basic information for the telemetry parameter like parameterID, mnemonic, raw value, EU converted value, (if defined in the database) status and S/C time value. The FaPaAnalysisParameter class performs the basic functions like data quality checking, sample rate checking etc. The FaPaAnalysisParameter passes the pointer to the parameter to the corresponding parameter group when parameter is considered okay for the product, otherwise it passes the null pointer.

-Refer to Offline Analysis - Analysis Parameters - Figure 3.2-7;

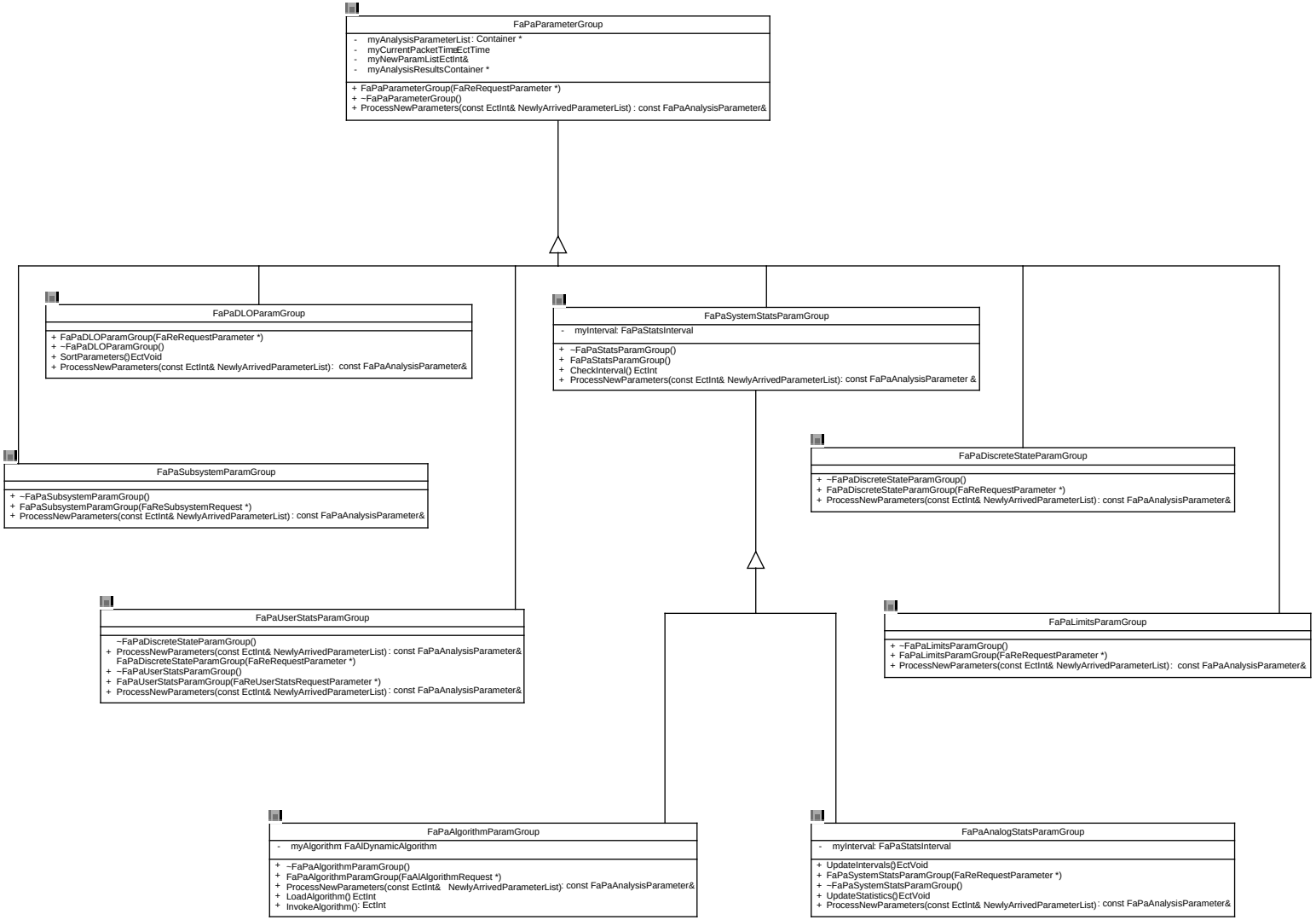

Figure 3.2-5. Offline Analysis Parameter Groups

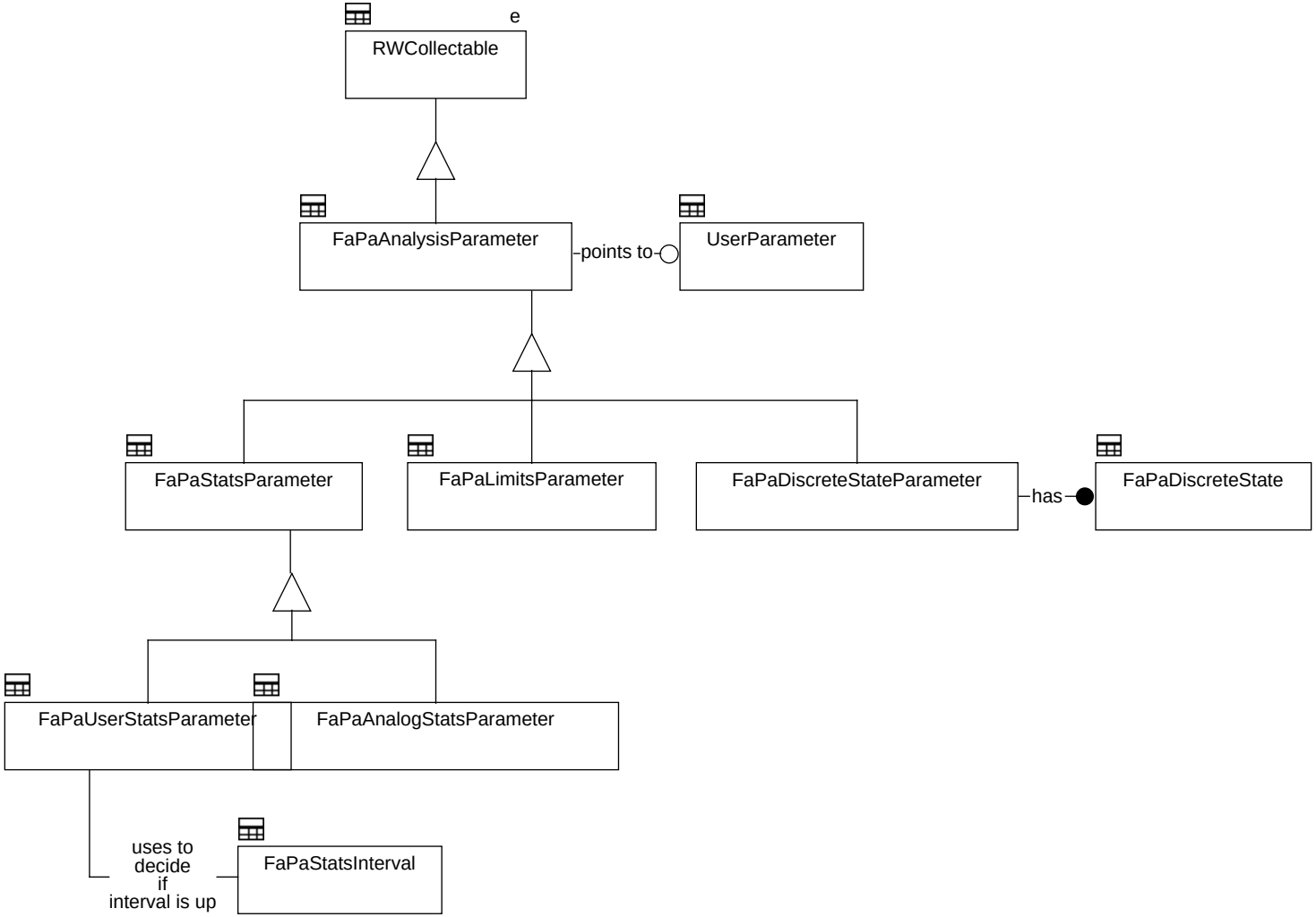

Figure 3.2-6. Offline Analysis Parameters

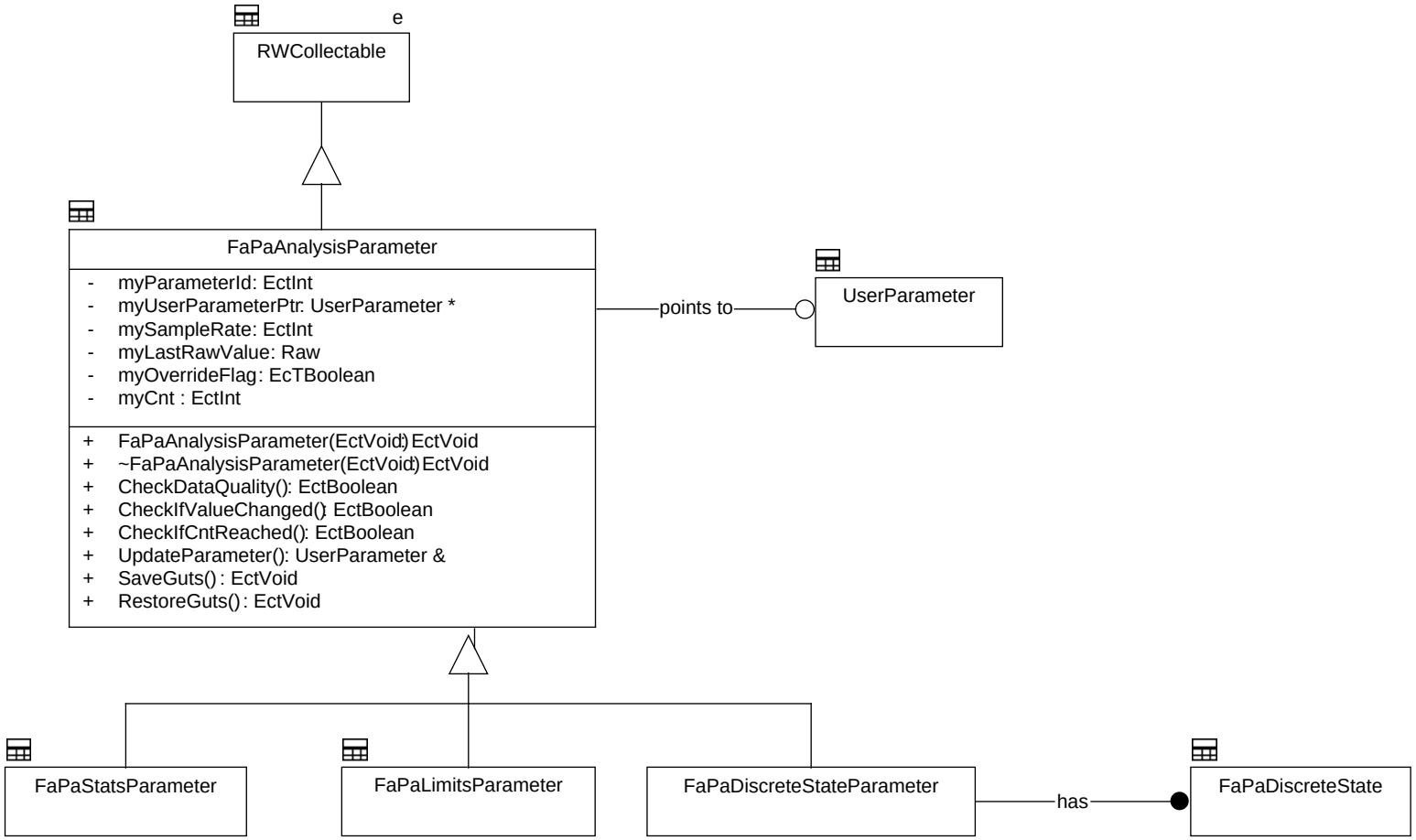

Figure 3.2-7. Offline Analysis - Analysis Parameters

3.2.3.7 Offline Analysis Limits and Discrete State Parameters

This model shows the processing of limits violation and discrete state change determinations. The FaPaLimitsParameter keeps track of the limits violations when it occurs and writes it to the predetermined dataset. The FaPaDiscreteStatsParameter determines the state changes and updates corresponding state change counter. The duration of each state is accumulated in the daily total and at the end of the day, the information about each state for each discrete parameter is written on the predetermined dataset.

-Refer to Offline Analysis Limits and Discrete State Parameters - Figure 3.2-8;

3.2.3.8 Offline Analysis Statistics Parameters

This model shows the processing of the systems generated statistics and the user defined statistics. The FaPaStatsParameter is the base class for the statistics. It keeps track of the minimum and maximum value of the parameter and also running counter for the sum of values for a given time interval. When the time interval expires, it computes the average, std deviation and sum of squares for a parameter. The statistics values are written to the predetermined datasets and counters are reset to zero. The FaPaUserStatsParameter uses user defined time interval and FaPaAnalogStatsParameter gets its direction from the FaPaAnalogParamGroup which uses systems time interval for the system generated statistics.

-Refer to Offline Analysis Statistics Parameters - Figure 3.2-9;

3.2.3.9 Offline Analysis Telemetry Request Products

This model shows the different classes involved in the generation of a product when the Telemetry request is executed. The FaAcOfflineAnaController is the main class which receives the Analysis request. Based on the Analysis request it instantiates FaPrDataset class which is inherited from the FaPrAnalysisProduct class. The FaPrAnalysisProduct class instantiates FaPaParameterGroup classes based on the Analysis request. Each of these FaPaParameterGroup class instantiates FaPaAnalysisParameter class as many as needed based on the parameter list associated with the parameter group. When the new parameters arrive, the FaAcOfflineAnaController notifies FaPrAnalysisProduct to update itself. The FaPrAnalysisProduct updates each FaPaParameterGroup objects which are updated by the FaPaAnalysisParameter objects. Upon receiving the pointers to the parameters, the FaPrAnalysisProduct (FaPrDataset) writes the parameter values to the dataset. When all the parameters are processed for a given time span, the FaPrAnalysisProduct, deletes all the FaPaParameterGroup objects which in turn deletes all the FaPaAnalysisParameter objects. The FaAcOfflineAnaController notifies Analysis Request Manager about completion of the request and the Analysis process is terminated.

-Refer to Offline Analysis Telemetry Request Product - Figure 3.2-10;

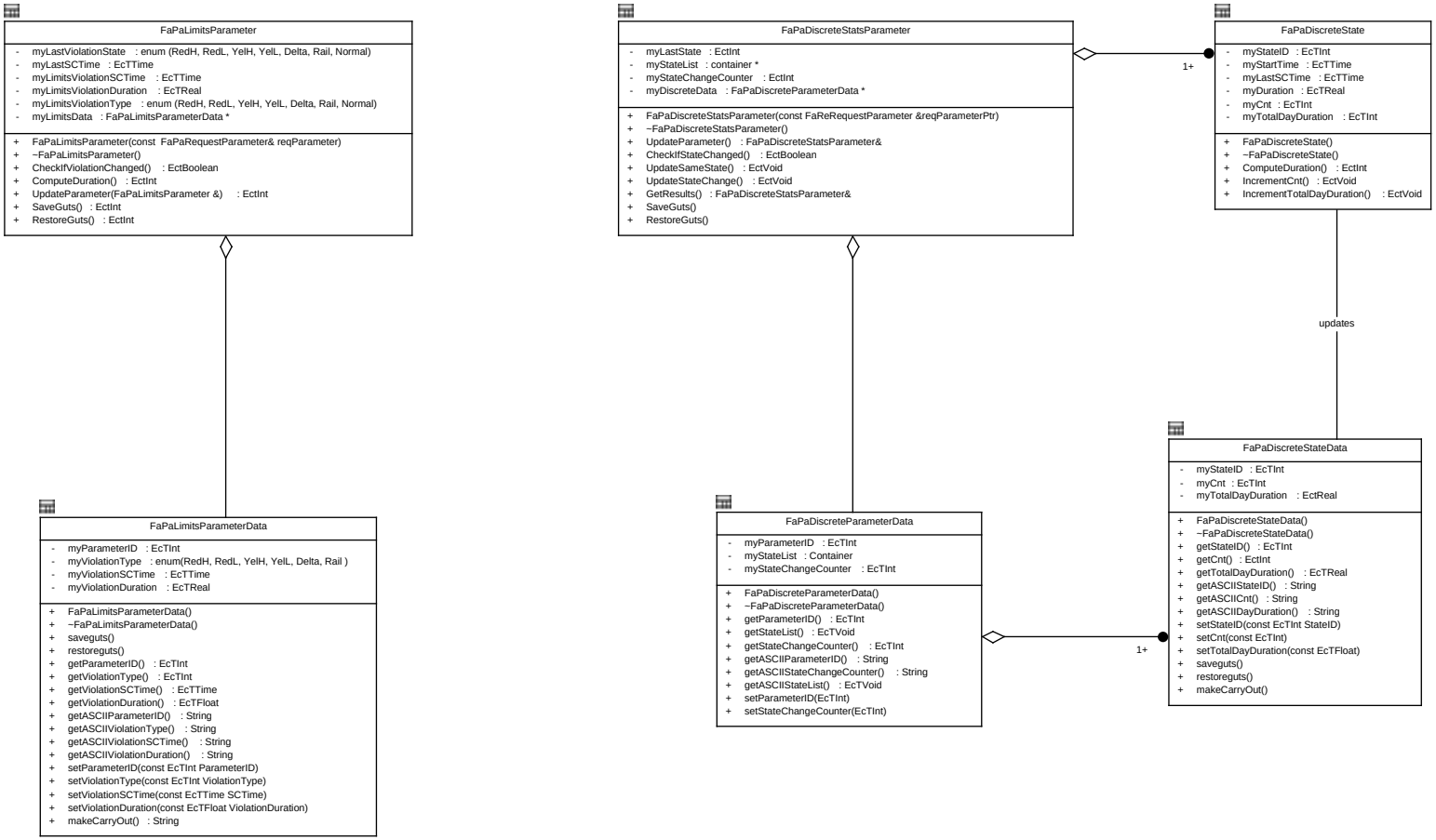

Figure 3.2-8. Offline Analysis Limits and Discrete State Parameters

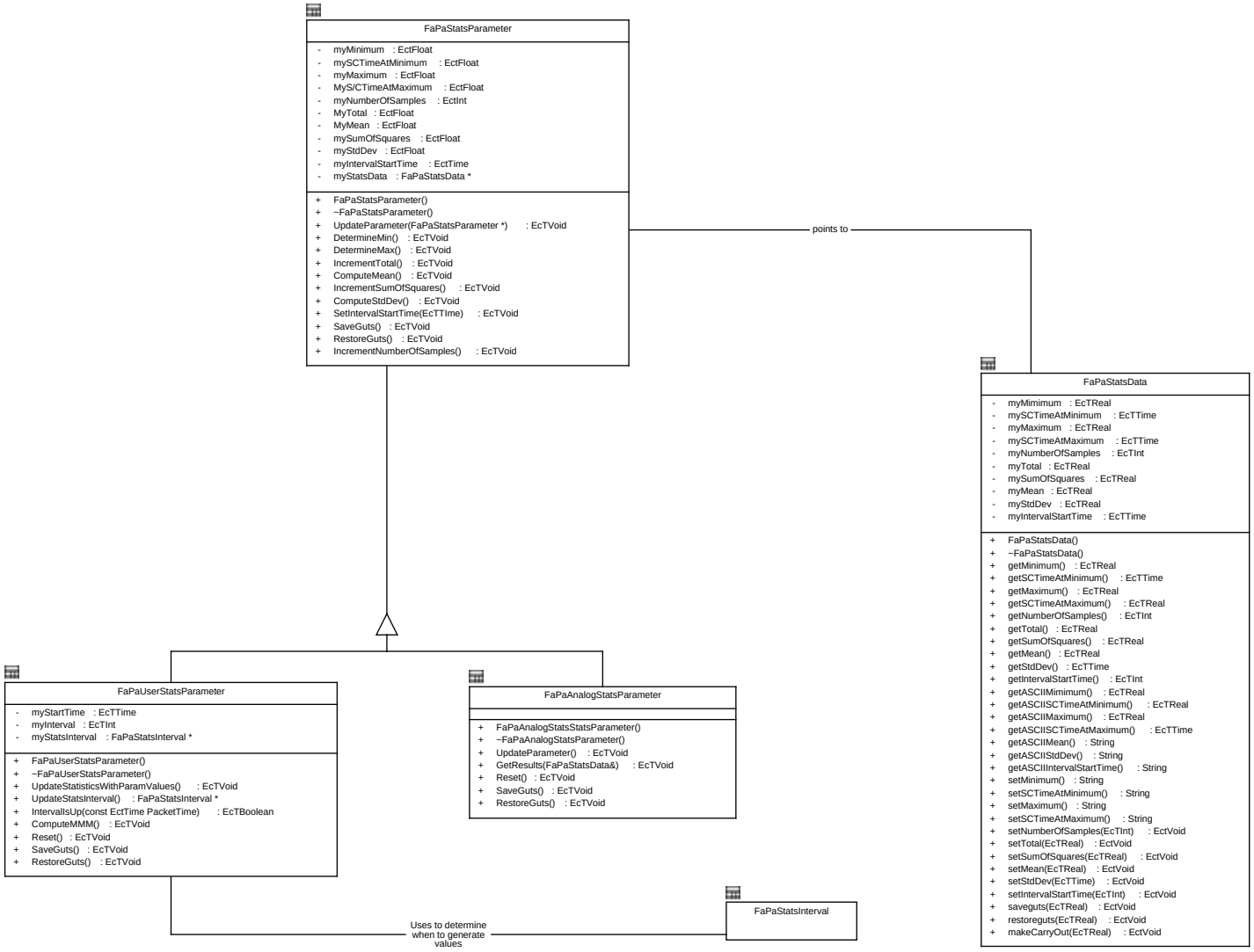

Figure 3.2-9. Offline Analysis Statistics Parameters

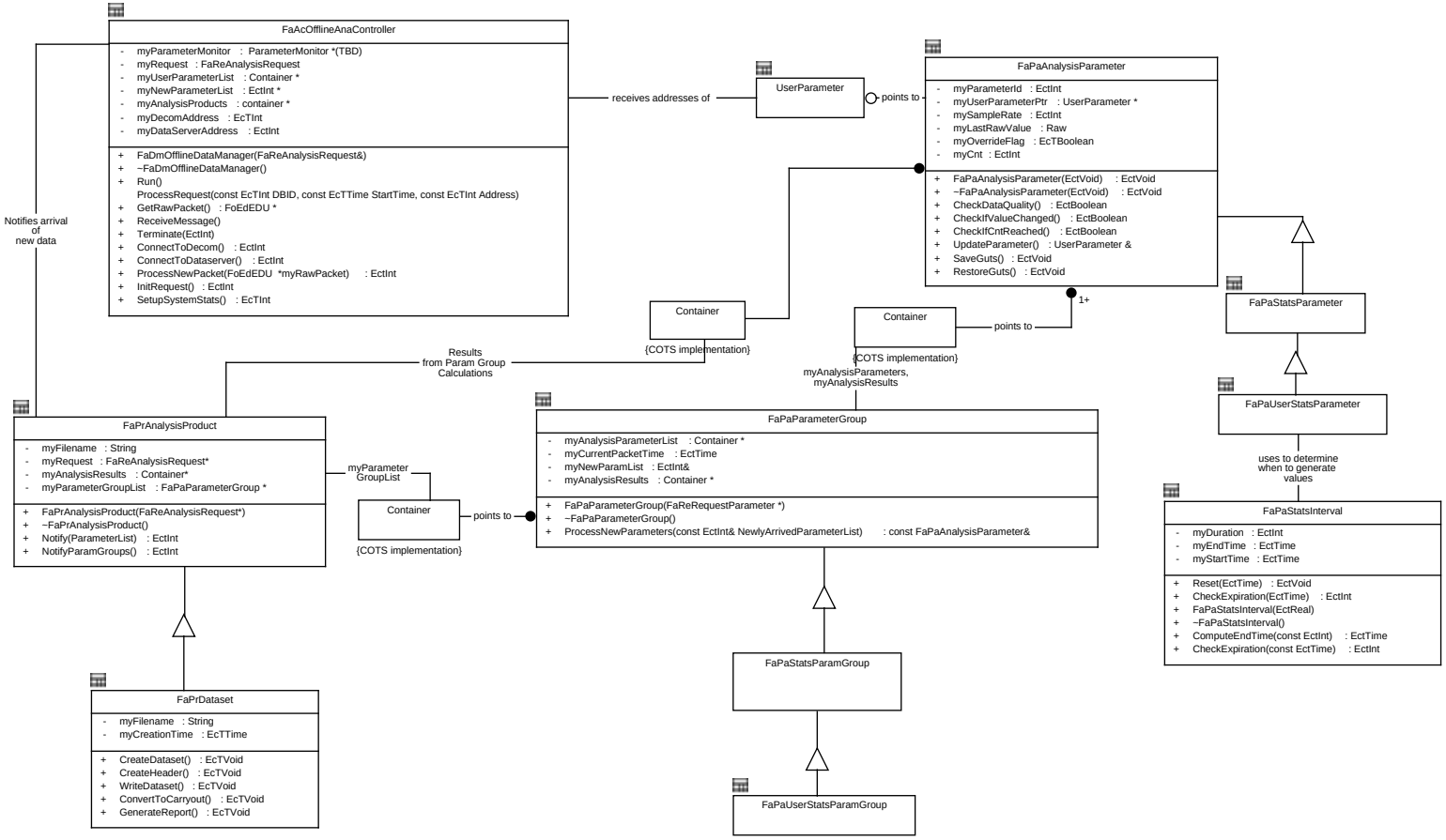

Figure 3.2-10. Offline Analysis Telemetry Request Product

3.2.3.10 Offline Analysis System Generated Statistics

This model depicts the processing involved in the systems generated statistics. The FaPaAnalogStatsParamGroup instantiates FaPaAnalogStatsParameter objects. The FaPaStatsInterval object determines when the monthly time interval is up. At that time FaPaSystemStatsDataset invokes saveguts function in each of the FaPaAnalogStatsParameter objects to write the monthly statistics on the dataset. The yearly and MTD statistics are generated from the monthly statistics.

-Refer to Offline Analysis System Generated Statistics - Figure 3.2-11;

3.2.3.11 Offline Analysis System Generated Limits Statistics

This model depicts the processing involved in the system generated limits statistics. The FaPaLimitsParameterGroup instantiates FaPaLimitParameter objects which keeps track of the limits violations and the duration's for each telemetry analog parameter. Each limit violation duration's and the violation types are written on the systems Limits dataset via FaPrLimitsDataset.

-Refer to Offline Analysis System Generated Limits Statistics - Figure 3.2-12;

3.2.3.12 Offline Analysis System Generated Discrete State Statistics

This model depicts the processing involved in the system generated discrete state statistics. The FaPaDiscreteStateParamGroup instantiates FaPaDiscreteStatsParameter objects for all discrete parameters. The FaPaDiscreteStatsParameter keeps track of state changes and the state duration. It updates the total daily duration counter for each state for each discrete parameter. At the end of each day, the FaPrDiscreteStateDataset writes the discrete state statistics for each parameter in the systems discrete dataset file.

-Refer to Offline Analysis System Generated Discrete Limits Statistics - Figure 3.2-13

3.2.3.13 Analysis Request

The Analysis request provides the interface between the User Interface Subsystem and the Analysis Subsystem. The User can specify collection of historical (replay) telemetry data using different sampling rates for the parameters and/or complicated algorithms to apply on the telemetry data. The systems generated statistics and user defined statistics are also obtained using Analysis Request.

This model shows different components of the Analysis request and the methods to fill in the fields or extract the values from the fields, from the request.

-Refer to Analysis Request object model - Figure 3.2-14

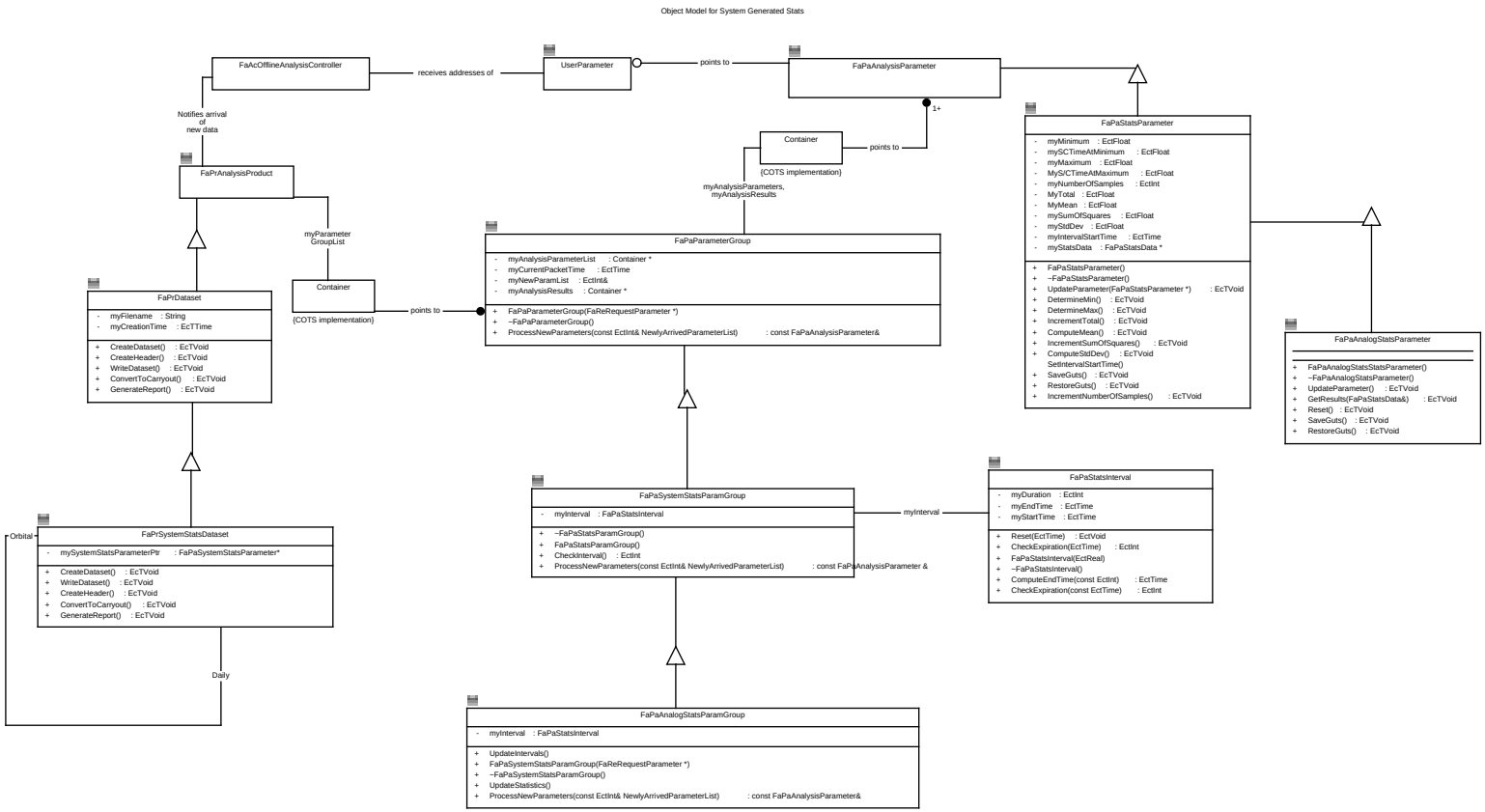

Figure 3.2-11. Offline Analysis System Generated Statistics

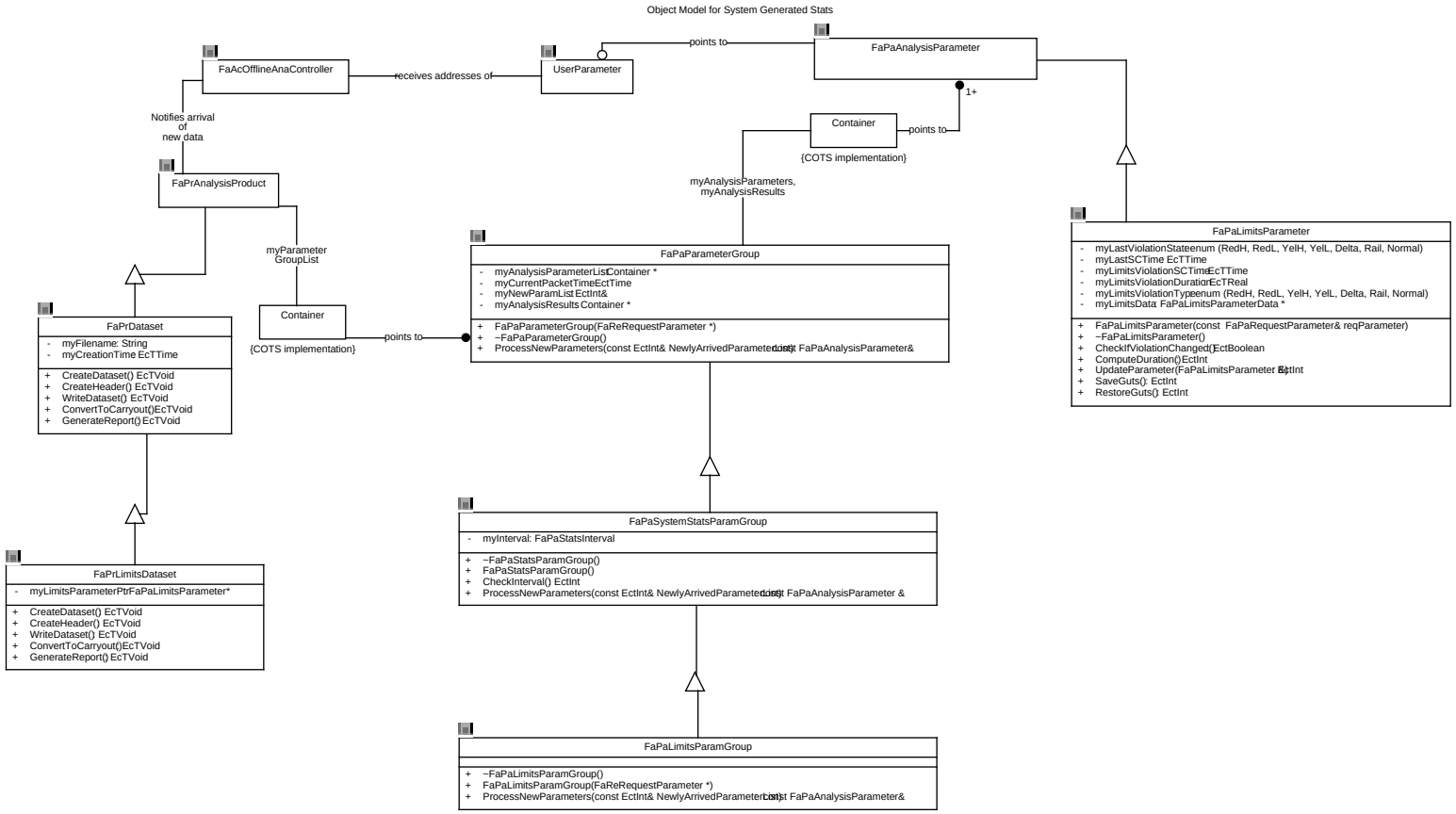

Figure 3.2-12. Offline Analysis System Generated Limits Statistics

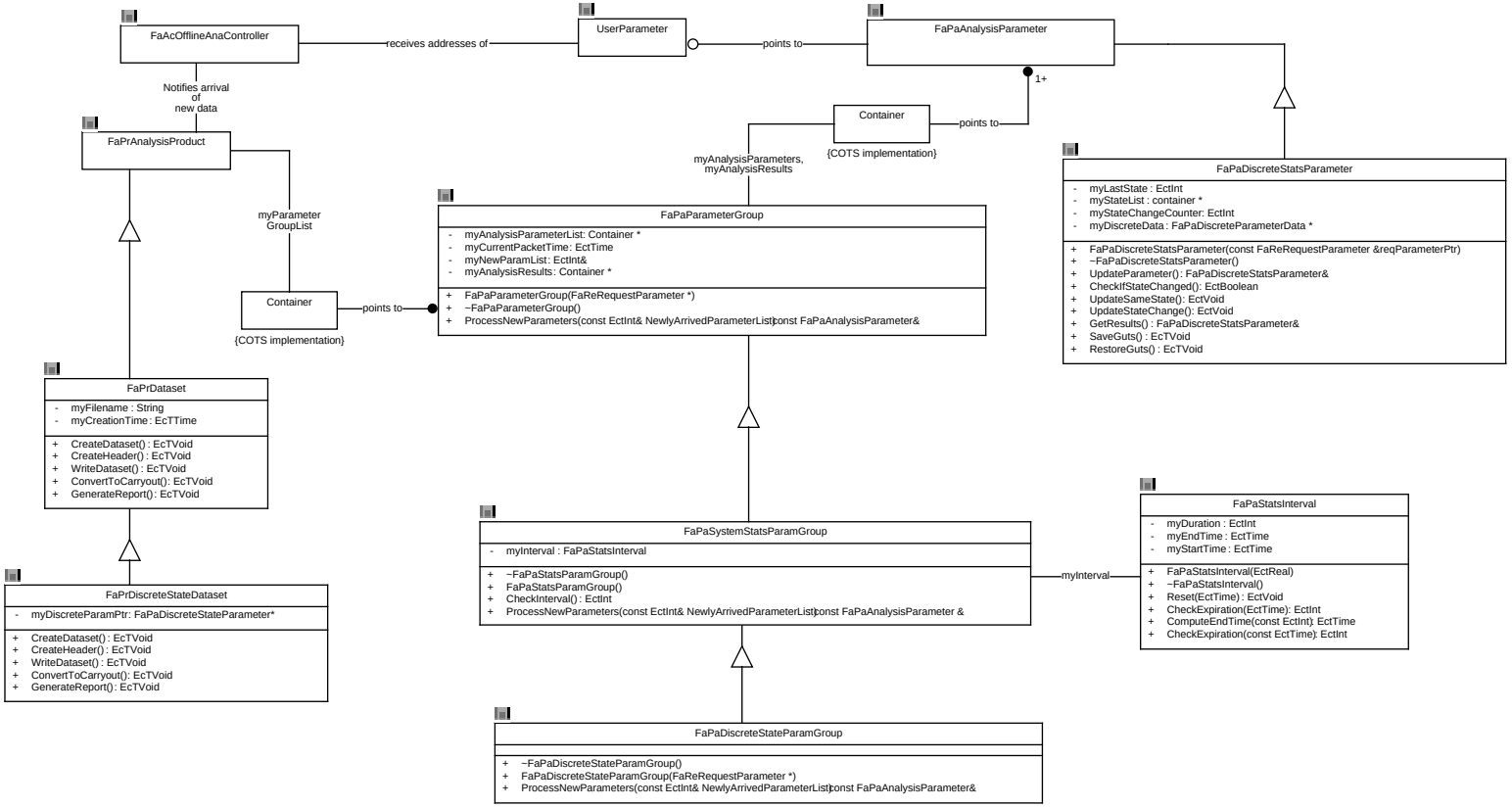

Figure 3.2-13. Offline Analysis System Generated Discrete Statistics

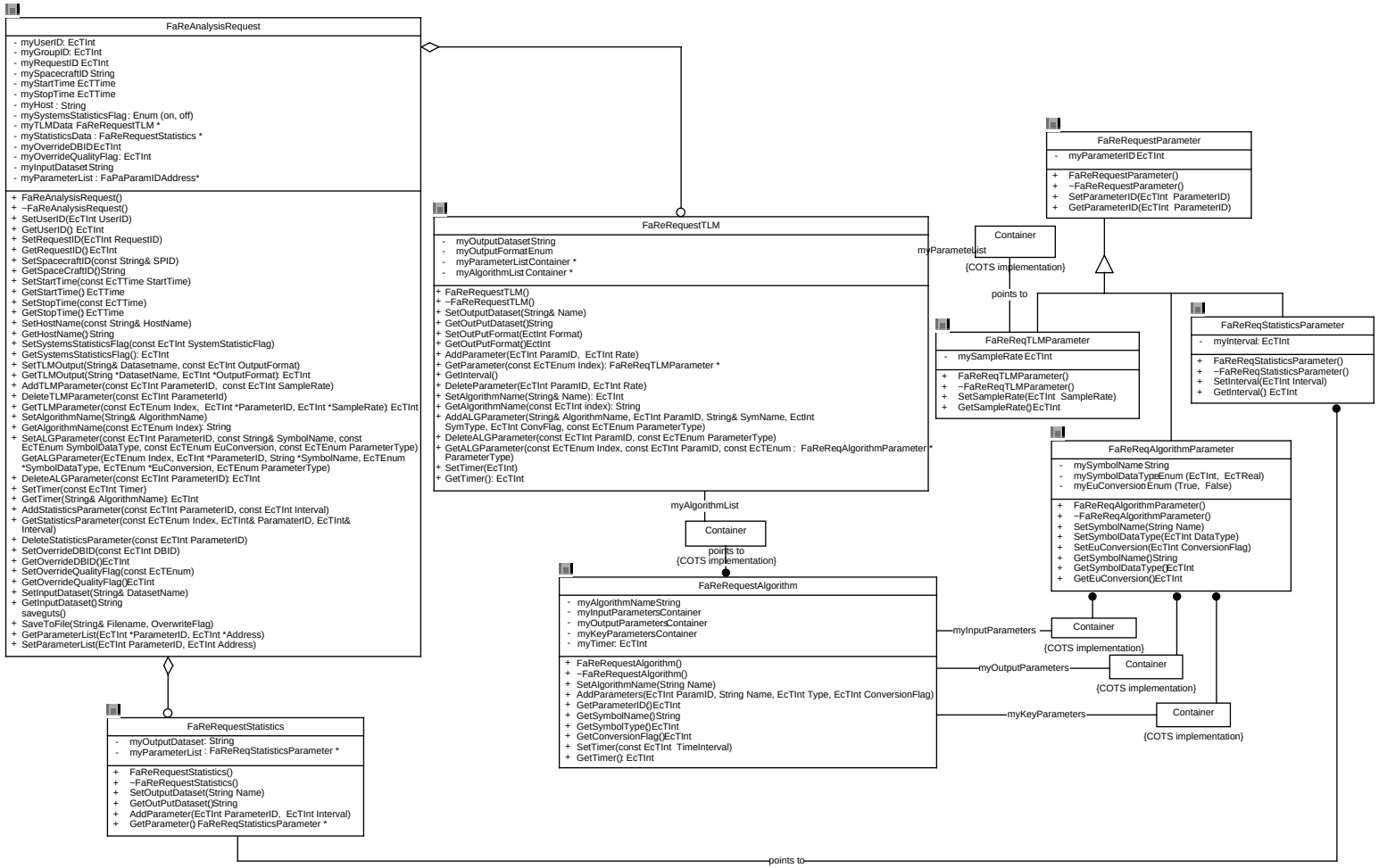

Figure 3.2-14. Analysis Request Object Model

3.2.4 Offline Analysis Dynamic Model

The Offline Analysis Dynamic model comprises several scenarios:

- Basic Dataset Generation
- System Generated Analog Statistics
- Daily Stats from Orbital Stats
- System Generated Limits Statistics
- System Generated Discrete Stats Statistics

3.2.4.1 Basic Dataset Generation Scenario

3.2.4.1.1 Basic Dataset Generation Scenario Abstract

The event trace diagram describes the complete scenario about the Basic Dataset Generation processing done by the Offline Analysis process. Telemetry datasets are created based on a user specified sampling rate which can be every sample, every Nth sample or change only.

Refer to Basic Dataset Generation Event Trace Figure 3.2-15

3.2.4.1.2 Basic Dataset Generation Summary Information

Interfaces:

- FOS Telemetry Subsystem
- FOS Data Management Subsystem

Stimulus:

When User submits the Analysis request to generate the Telemetry Dataset, the Analysis Request Manager receives Analysis request from the User Interface subsystem. The Analysis Request Manager starts up the Offline Analysis process for each Analysis Request received.

Desired Response:

When the Analysis request processing is completed, the Offline Analysis process sends the completion status to the Analysis Request Manager which in turn, sends the completion status to the User Interface subsystem.

Pre-Conditions: None

Post-Conditions: None

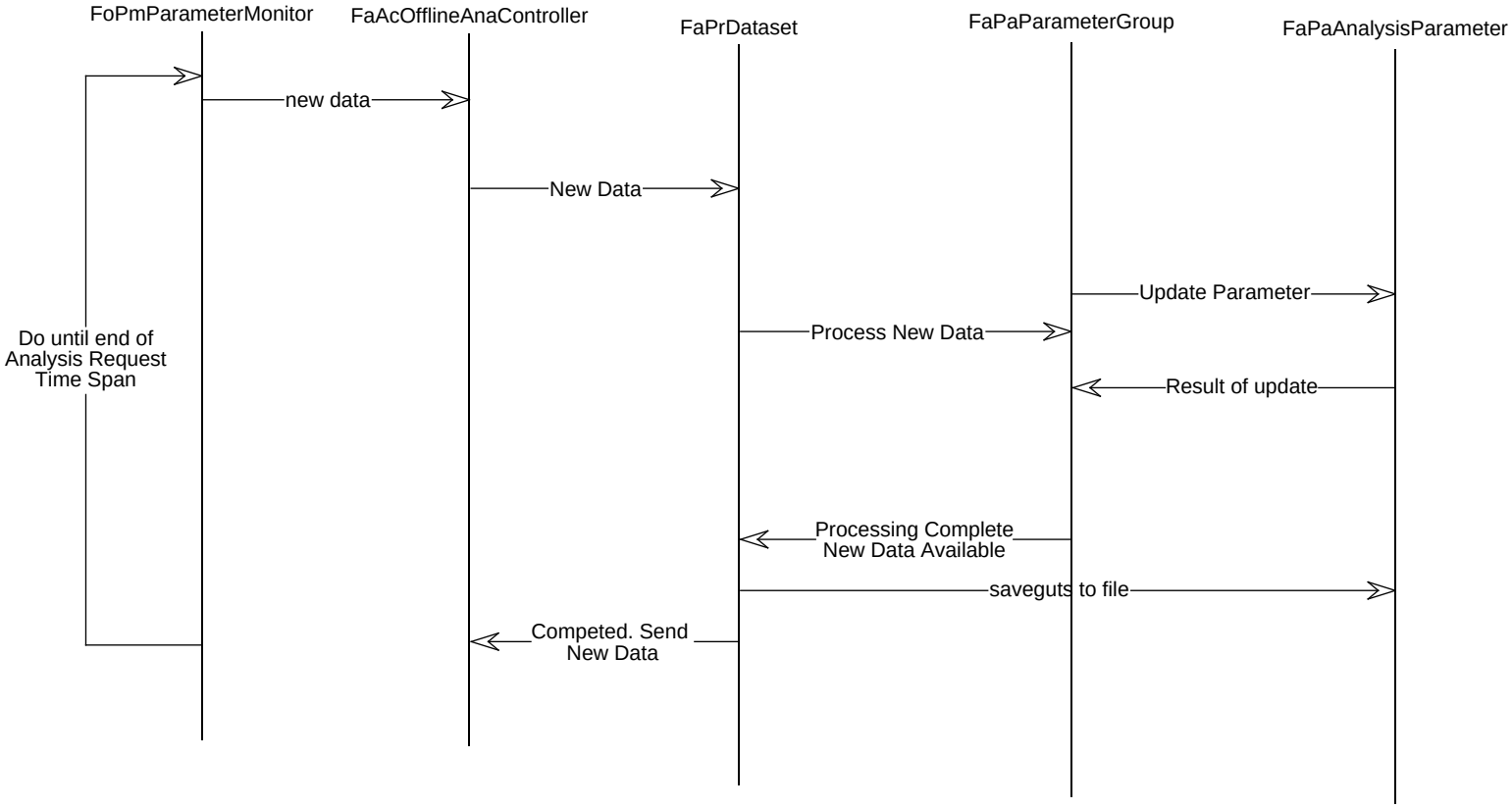

Figure 3.2-15. Basic Dataset Generation Event Trace

3.2.4.1.3 Basic Dataset Generation Scenario Description

Data flows from TLM to the Offline FaAcOfflineAnaController. The FaAcOfflineAnaController notifies all products (usually just one FaPaDataset) that new telemetry is available. The FaPaDataset will notify it's FaPaParameterGroups of the new data, and each group will do whatever processing is required. In the case of a simple telemetry dataset, parameters are checked to see if the user selected sample rate indicates the parameter should be stored in the dataset, and those parameters which are ready are returned to the FaPaDataset to be saved. Another parameter group associated with the Telemetry Dataset is the FaPaAlgorithm, which is responsible for applying User supplied functions to the telemetry data and returning the results to the dataset.

3.2.4.2 System Generated Analog Statistics Scenario

3.2.4.2.1 System Generated Analog Statistics Scenario Abstract

Analog statistics are composed of the Minimum, Time of the Minimum, Maximum, Time of the Maximum, Mean and Standard Deviation for the following time intervals: Orbit Night, Orbit Day, Full Orbit, Calendar Day, Month, and mission to date. They are performed automatically upon receipt of back orbit telemetry, for all analog parameters and analog derived parameters.

Reference System Generated Analog Statistics Event Trace Figure 3.2-16

3.2.4.2.2 System Generated Analog Statistics Scenario Summary Information

Interfaces:

- FOS Telemetry Subsystem
- FOS Data Management Subsystem

Stimulus: The Analysis Request is submitted automatically upon receipt of back orbit telemetry, for all analog parameters and analog derived parameters.

Desired Response: The following system statistics datasets are generated and stored in the systems files: Orbit Night, Orbit Day, Full Orbit, Calendar Day, Month, and mission to date.

Note: All these datasets are generated at the appropriate time. For example, statistics for a Month is generated when one month worth of data is accumulated.

Pre-Conditions: None

Post-Conditions: None

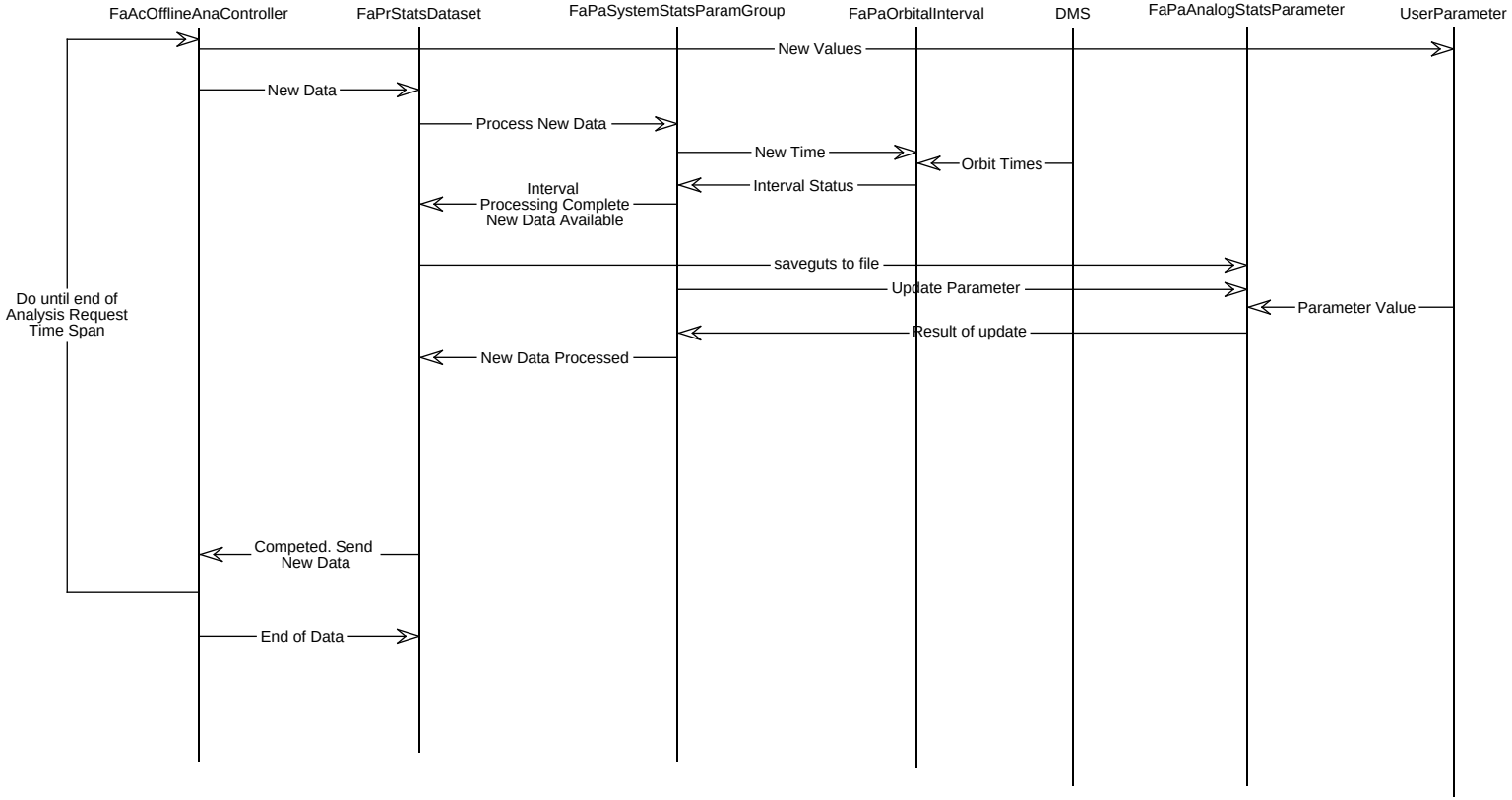

Figure 3.2-16. System Generated Analog Statistics Event Trace

3.2.4.2.3 System Generated Analog Statistics Scenario Description

Data flows from TLM to the FaAcOfflineAnaController. The FaPaOfflineAnalysisController notifies the FaPaOrbitStatsDataset that new telemetry is available. The FaPaOrbitStatsDataset will notify it's FaPaAnalogStatsParamGroup of the new data, and the parameter group will first check to see if the current orbital interval has expired. If it has expired, the parameter group will ask all of it's FaPaAnalogStatsParameters for their values, then tell the FaPaAnalogStatsParameters with new telemetry values to update. Values of all stats parameters are returned to the FaPaOrbitStatsDataset if a new Orbit interval has been reached.

After all data is processed, a FaPaDailyStatsDataset is instantiated, and if a new calendar daily interval has been reached, the Orbit stats are used to generate a new entry in the FaPaDailyStatsDataset.

3.2.4.3 Daily Stats from Orbital Stats Scenario

3.2.4.3.1 Daily Stats from Orbital Stats Scenario Abstract

This scenario will show how daily interval system statistics are generated from orbital interval statistics.

Reference Figure 3.2-17 Daily Stats from Orbital Stats Event Trace

3.2.4.3.2 Daily Stats from Orbital Stats Scenario Summary Information

Interfaces:

- FOS Telemetry Subsystem
- FOS Data Management Subsystem

Stimulus: Orbital stats are automatically generated and completed, and the final orbit ended within the time span of the next GMT calendar day.

Pre-Conditions: Orbital stats have been generated on the current Day of Year's data.

Post-Conditions: Daily stats are now available for larger statistical intervals.

3.2.4.3.3 Daily Stats from Orbital Stats Scenario Description

Upon the completion of a full day of orbital statistics, the FaAcOfflineAnaController creates an instance of a FaPaStatsDataset for a daily interval, and begins to read data from the FaPaOrbitStatsDataset, causing the update of FaPaSystemStatsParameters with Orbit values. The FaPaSystemStatsParameters for daily values use the orbit values for the entire day to build stats for the daily interval. The FaPaDailyInterval is used to determine when the current day interval has expired. Then the FaPaStatsDataset instructs it's FaPaSystemStatsParameters to save themselves to the dataset file. The FaAcOfflineAnalysisController is then informed of the completion of processing.

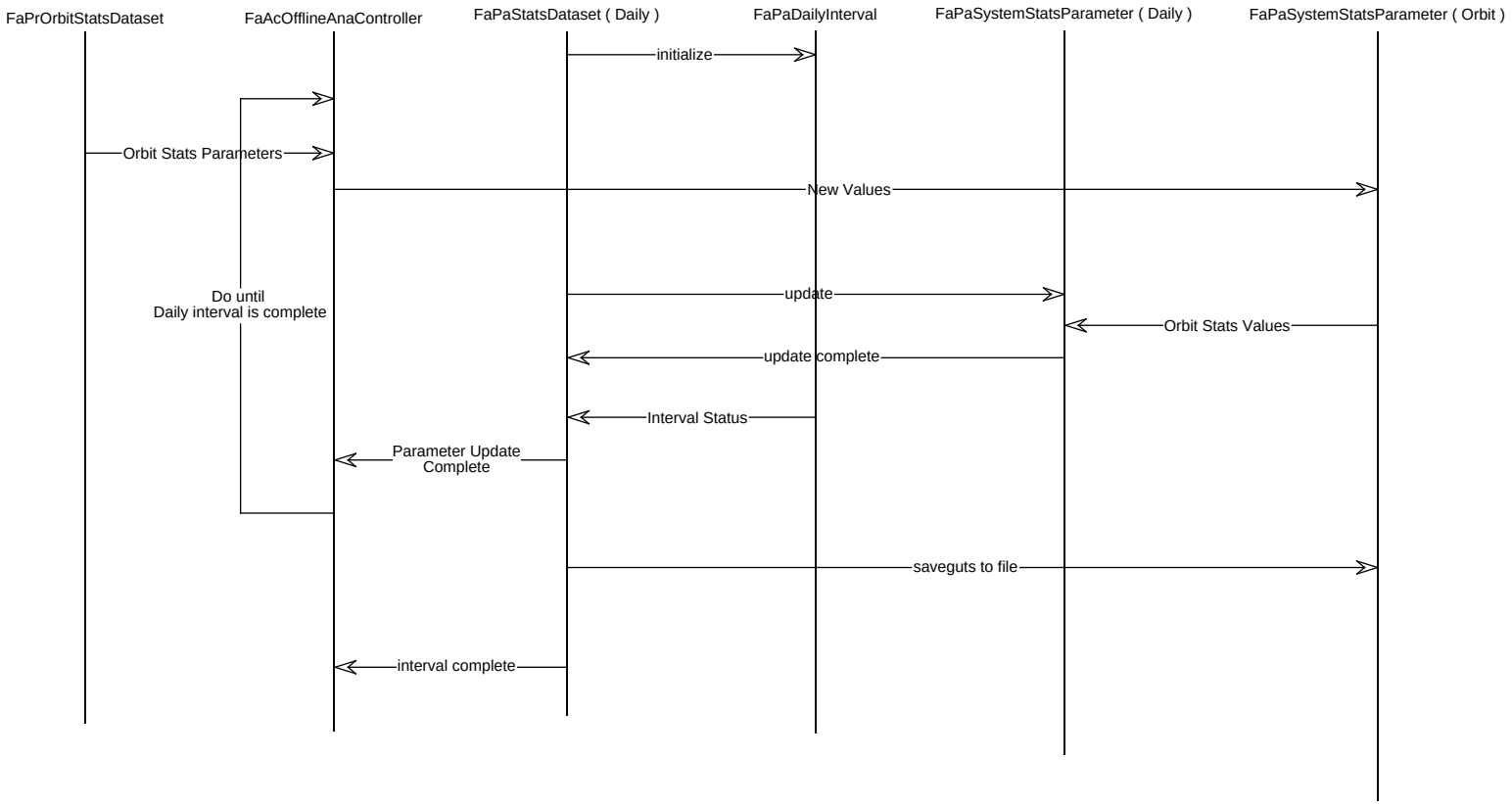

Figure 3.2-17. Daily Stats from Orbital Stats Event Trace

3.2.4.4 System Generated Limits Statistics Scenario

3.2.4.4.1 System Generated Limits Statistics Scenario Abstract

Limits statistics consists of the following information for each limit violation. These are kept for the life of the mission.

a) Limit type (Red/Yellow, High/Low).

b) Limit start time.

c) Duration of violation.

Reference Figure 3.2-18 System Generated Limits Statistics Event Trace

3.2.4.4.2 System Generated Limits Statistics Scenario Summary Information

Interfaces:

- FOS Telemetry Subsystem
- FOS Data Management Subsystem

Stimulus: System Generated Limits Statistics are generated along with the System Generated Analog Statistics.

Desired Response: None

Pre-Conditions: None

Post-Conditions: None

3.2.4.4.3 System Generated Limits Statistics Scenario Description

Data flows from TLM to the Offline Analysis Controller (FaAcOfflineAnaController). The Offline Analysis Controller notifies the FaPaLimitsDataset that new telemetry is available. The FaPaLimitsDataset will notify it's FaPaLimitsParamGroup of the new data. The parameter group will then notify all FaPaLimitsParameters that have new telemetry values. Each parameter notified will check to see if there is a new limit violation, if a limit violation has ended, or if a limit violation has changed. If a limit violation has changed, or if it is over, then the value of the FaPaLimitsParameter is returned to the FaPaLimitsDataset to be saved.

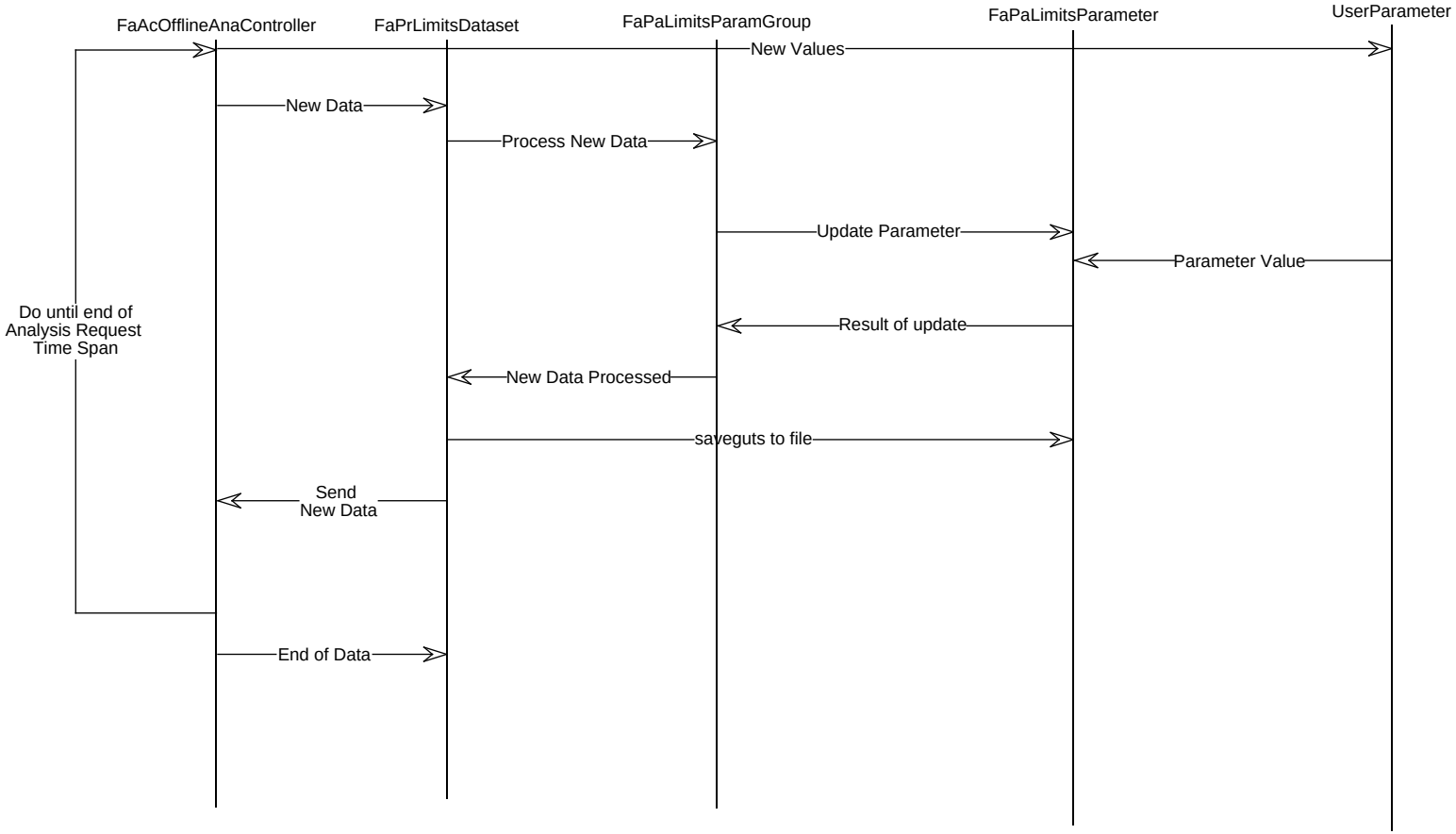

Figure 3.2-18. System Generated Limits Statistics Event Trace

3.2.4.5 System Generated Discrete Statistics Scenario

3.2.4.5.1 System Generated Discrete Statistics Scenario Abstract

Discrete statistics consist of the total number of state changes, and the total duration in each state, for the intervals of calendar day, month, and mission to date. They are performed on all discrete parameters and any derived parameters of a discrete nature.

Reference Figure 3.2-19 System Generated Discrete Statistics Event Trace

3.2.4.5.2 System Generated Discrete Statistics Scenario Summary Information

Interfaces:

- FOS Telemetry Subsystem
- FOS Data Management Subsystem

Stimulus: System Generated Discrete Statistics are generated along with the System Generated Analog Statistics.

Desired Response: Daily, monthly, and mission-to-date discrete statistics files are generated.

Pre-Conditions: None

Post-Conditions: None

3.2.4.5.3 System Generated Discrete Statistics Scenario Description

This scenario is similar to 3.4.1.2.2, except that only calendar day and calendar month intervals are used for discrete state statistics, and the resulting data are FaPaDiscreteStatsParameters.

Data flows from TLM to the FaAcOfflineAnaController. The FaPaOfflineAnalysisController notifies the FaPaOrbitStatsDataset that new telemetry is available. The FaPaOrbitStatsDataset will notify it's FaPaAnalogStatsParamGroup of the new data, and the parameter group will first check to see if the current orbital interval has expired. If it has expired, the parameter group will ask all of it's FaPaAnalogStatsParameters for their values, then tell the FaPaAnalogStatsParameters with new telemetry values to update. Values of all stats parameters are returned to the FaPaOrbitStatsDataset if a new Orbit interval has been reached.

After all data is processed, a FaPaDailyStatsDataset is instantiated, and if a new calendar daily interval has been reached, the Orbit stats are used to generate a new entry in the FaPaDailyStatsDataset.

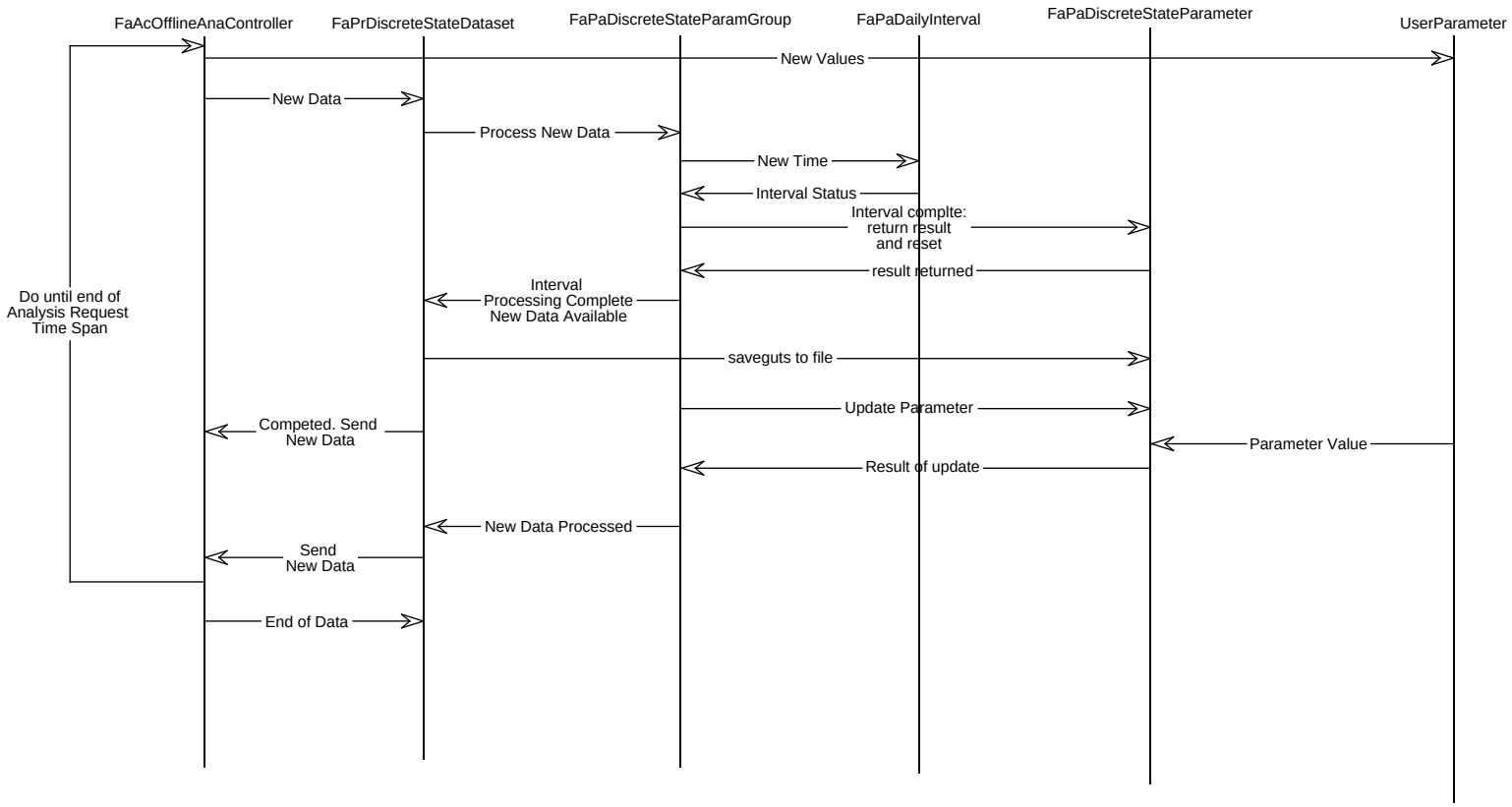

Figure 3.2-19. System Generated Discrete Statistics Event Trace

3.2.5 Offline Analysis Data Dictionary

FaDmOfflineDataManager

class **FaDmOfflineDataManager**

class FaDmOfflineDatamanager

This class controls the analysis data processing by getting packets from DMS, sending them to Decom and then notifying all FaPaAnalysisProducts that updated data values are available

Public Construction

FaDmOfflineDataManager(FaReAnalysisRequest)

constructor assigns request to myRequest

~FaDmOfflineDataManager(void)

destructor

Public Functions

EctInt **ConnectToDataserver**(void)

ConnectToDataServer

Connects to data server for raw tlm packets

EctInt **ConnectToDecom**(void)

Connects to the decom process which is feeding UserParameters to the FaDmOfflineDataManager

FoEdEDU **GetRawPacket**()

Get raw packet from DataServer

EctInt **InitRequest**(void)

This function will instantiate all the FaPrAnalysisProducts needed to perform the request

EctInt **ProcessNewPacket**(FoEdEDU& myRawPacket)

When a new packet of user parameters is received, this function is called to tell all products requiring the data that it is available

EctInt **ProcessRequest**(const EctInt DBID, const EcTTime StartTime, const EcTInt Address)

Processes request corresponding to each DBID

EcTVoid **ReceiveMessage**()

Upon receiving the abort message, this member function terminates the Analysis Request processing.

EcTVoid **Run**()

This member function initiates the connections and starts processing of the request. When all data is processed in the specified time, it terminates the process

EctInt **SetupSystemsStats**(void)

SetupSystemStats

This function will instantiates all the objects needed for the Systems Generated Statistics

void **Terminate**(EctInt)

Terminate.

Closes all connections and notifies all products that end of data has been reached

Private Data

`EcTInt` **`myDECOMaddress`**

This member variable stores the DECOM process address which is received from Analysis Request Manager process

`EcTInt` **`myDataServeraddress`**

This member variable stores the DataServer process address which is received from Analysis Request Manager process

`EctInt*` **`myNewParameterList`**

myAnalysisProducts

a container of AnalysisProducts. These are the classes which use the received UserParameters to generate a product

`FaReAnalysisRequest` **`myRequest`**

A copy of the analysis request specifying what products to generate

`Container*` **`myUserParameterList`**

A container of user parameters, updated with values received from decom

FaPaAlgorithmParamGroup

class **FaPaAlgorithmParamGroup**

Base Classes

`public` **`FaPaParameterGroup`**

Public Construction

`FaPaAlgorithmParamGroup``(FaAlAlgorithmRequest)`
`~FaPaAlgorithmParamGroup``(void)`

Public Functions

`EctInt` **`InvokeAlgorithm`**`(void)`

Binds the values of the parameters needed by the algorithm to the algorithms local data. then invokes the functions call

`EctInt` **`LoadAlgorithm`**`(void)`

Uses dynamic link library functions to load a previously built shared object file, and link parameters within the file to parameters with in the FaPaAlgorithmParamGroup

`const FaPaAnalysisParameter&` **`ProcessNewParameters`**`(const EctInt& NewlyArrivedParameterList)`

Determines if the algorithm should be invoked by looking for updates of key parameters, or looking at the elapsed time since the last invocation.

Private Data

`FaAlDynamicAlgorithm` **`myAlgorithm`**

This is the class used to encapsulate the algorithm functions such as dynamic linking and invocation.

FaPaSystemStatsParamGroup

class **FaPaSystemStatsParamGroup**

Class FaPASystemStatsParamGroup

This param group has a container of FaPaAnalogStatsParameters for which orbital stats are generated. The orbital interval is determined by FaPaOrbitStatsInterval, which uses FDF data to determine orbit and eclipse times

Base Classes

public **FaPaSystemStatsParamGroup**

Public Construction

FaPaSystemStatsParamGroup(FaReStatsRequestParameter*)
~FaPaSystemStatsParamGroup(void)

Public Functions

const FaPaAnalysisParameter& **ProcessNewParameters**(const EctInt& NewlyArrivedParameterList)

First checks if interval has expired and saves statistical results if it has, then updates all parameters which have new values.

void **UpdateStatistics**(void)

all statistics parameters whose user parameters have updated are processed to reflect the new data values.

Private Data

myInterval **FaPaOrbitStatsInterval**

Used to update statistics based on orbit and orbit day and night duration's

FaPaAnalogStatsParameter

class **FaPaAnalogStatsParameter**

This class Updates Analog Systems Statistics parameter with the new data. Upon receiving compute statistics command from the parameter group, it computes statistics for the parameter and stores into a separate object. Upon receiving write directive from the Analysis Product, it writes parameter statistics to a dataset file.

Base Classes

public **FaPaStatsParam**

Public Construction

FaPaAnalogStatsParameter(void)

This member function is a default constructor for the FaPaAnalogStatsParameter class

~FaPaAnalogStatsParameter(void)

This member function is a default destructor for the FaPaAnalogStatsParameter class

Public Functions

FaPaAnalogStatsParameter& **ComputeStatistics**(void)

This member function computes statistical values like Mean, Standard Deviation etc.

void **Reset**(void)

This member function initializes various attributes to zero

void **RestoreGuts**()

This member function reads data from a dataset file using input stream

void **SaveGuts**()

This member function writes to a dataset file using output stream

void **UpdateParameter**()

This member function updates statistical data with the new data.

FaPaAnalysisParameter

class **FaPaAnalysisParameter**

This class is a base class for the analysis parameters. It contains attributes and operations necessary for getting the parameter values (EU converted and/or raw) for the analysis products. NOTE: Since this class points to a UserParameter class

```
(common parameter class for an User Interface Subsystem
and an Analysis Subsystem) please refer to  UserParameter
class  for further details
```

Base Classes

public **RWcollectable**

Public Functions

EcTBoolean **CheckDataQuality**(EcTVoid)

This member function checks the data quality of the analysis parameter by checking status attribute value

EcTBoolean **CheckIfCntReached**(EcTVoid)

This member function checks if the sample counter has reached an user specified number. In certain analysis products user may be interested in every nth sample rather than all samples

EcTBoolean **CheckIfValueChanged**(EcTVoid)

This member function checks if the current value is different from the previously stored value. For certain analysis products, User may be interested in "changed only" values

EcTVoid **FaPaAnalysisParameter**(EcTVoid)

This member function is a default constructor for the FaPaAnalysisParameter

EcTInt **SaveGuts**(void)

This member function writes the attributes to the output stream

UserParameter* **UpdateParameter**(EcTVoid)

This member function returns the pointer to a user parameter after checking data quality and sampling rate

Private Data

EcTInt **myCnt**

This member variable gives the counter for sampling rate

Raw **myLastRawValue**

This member variable stores the previous raw value of a parameter

EcTBoolean **myOverrideFlag**

This member variable indicates if the bad quality data value should be considered for the analysis product or not

EcTInt **myParameterId**

This member variable is a parameter identifier

EcTInt **mySampleRate**

This member variable provides the rate at which the parameter is collected in the product. For the change only sampling rate the parameter is considered in a product only if it's value changes from the previous one.

UserParameter* **myUserParameterPtr**

This member variable points to a User parameter

FaPaDLOParamGroup

class **FaPaDLOParamGroup**

This class is exactly the same as the FaPaPamameterGroup, except it also has the ability to sort parameters in time order, so

they can be printed in a Downlink Ordered Report.

Base Classes

`public` **`FaPaParameterGroup`**

Public Construction

`FaPaDLOParamGroup``(FaReRequestParameter*)`

constructor

`~FaPaDLOParamGroup``(void)`

destructor

Public Functions

`void` **`SortParameters`**`(void)`

This function will sort the parameters in myAnalysisResults before they are made available to the parent product.

FaPaDiscreteParameterData

`class` **`FaPaDiscreteParameterData`**

This class represents the information related to the Discrete Parameter data. This object is used to read and write stream I/O for Discrete Parameter information

Public Construction

`FaPaDiscreteParameterData``(void)`

This member function is a default constructor for

`FaPaDiscreteParameterData class`

`~FaPaDiscreteParameterData``(void)`

This member function is a default destructor for

`FaPaDiscreteParameterData class`

Public Functions

```
String getASCIIParameterID(void)
String getASCIIStateChangeCounter(void)
EcTVoid getASCIIStateList(void)
EcTInt getParameterID(void)
EcTInt getStateChangeCounter(void)
EcTVoid getStateList(void)
void setParameterID(EcTInt)
void setStateChangeCounter(EcTInt)
```

Private Data

`EcTInt` **`myParameterID`**

This member variable stores the parameter Id

`EcTInt` **`myStateChangeCounter`**

This member variable holds the count of my state change

`Container` **`myStateList`**

This member variable is a container for all the states associated with this parameter

FaPaDiscreteState

class **FaPaDiscreteState**

This class represents the information related to the state of the discrete parameter.

Public Construction

FaPaDiscreteState(void)

This member function is a default constructor for the FaPaDiscreteState class

~FaPaDiscreteState(void)

This member function is a default destructor for the FaPaDiscreteState class

Public Functions

void **ComputeDuration**(void)

This member function computes the duration of the state that

parameter was in.

void **IncrementCnt**(void)

This member function counts the number of instances that the discrete parameter had that state in a day

void **IncrementTotalDayDuration**(void)

This member function adds each new duration into the TotalDayDuration.

void **SaveGuts**(const EctInt FileDescriptor)

This member function writes each occurrence of a state, its start time, and the duration for the discrete parameter

Private Data

parameter **is**

This member variable gives the latest time of a state that the

EctInt **myCnt**

This member variable is a running counter for the occurrences of that state in a given day

EctFloat **myDuration**

This member variable gives the duration of a state that the parameter was in, since the start of that state.

EctTime **myStatrTime**

This member variable gives the start time of a state

EctInt **myTotalDayDuration**

This member variable provides the total duration of a particular state in a day.

FaPaDiscreteStateData

class **FaPaDiscreteStateData**

This class provides the unit information for a discrete state of an parameter

Public Construction

```
FaPaDiscreteStateData(void)
~FaPaDiscreteStateData(void)
```

Public Functions

```
String getASCIICnt(void)
String getASCIIDayDuration(void)
String getASCIIStateID(void)
EctInt getCnt(void)
EcTInt getStateID(void)
EcTFloat getTotalDayDuration(void)
void restoreguts(void)
void saveguts(void)
void setCnt(const EcTInt)
void setStateID(const EcTInt StateID)
void setTotalDayDuration(const EcTFloat)
```

`void makeCarryOut(void)`

This function returns a string of data corresponding to stateID, statecnt, and stateTotalDayDuration

Private Data

`EcTInt myCnt`

This member variable holds count for the occurrence of this state in a day

`EcTInt myStateID`

This member variable holds the state ID.

`EctFloat myTotalDayDuration`

This member variable holds the Total duration of the state in a given day

FaPaDiscreteStateParamGroup

class FaPaDiscreteStateParamGroup

This class is used when system statistics are generated. It monitors state changes of discrete state parameters and accumulates the values in a list of FaPaDiscreteStateParameters. For each parameter, the number of state changes and the total time spent in each state is kept for the day, month, and MTD.

Base Classes

`public FaPaSystemStatsParameterGroup`

Public Construction

`FaPaDiscreteStateParamGroup(FaReRequestParameter*)`

constructor This function will, for all discrete parameters, instantiate a

`FaPaDiscreteStateParameter.`

`~FaPaDiscreteStateParamGroup(void)`

destructor

Public Functions

```
const FaPaAnalysisParameter& ProcessNewParameters(const EctInt&
      NewlyArrivedParameterList)
```

This function will be invoked whenever new data is made available. It will cause each FaPaDiscreteStateParameter to update it's state change information based on the new data.

FaPaDiscreteStatsParameter

class **FaPaDiscreteStatsParameter**

This class is a derived class from the analysis parameter. It contains attributes and operations necessary to capture the different states parameter exhibits during the spacecraft operations.

Public Construction

FaPaDiscreteStatsParameter(FaReRequestParameter& reqParameterPtr)

This member function is a default constructor for the FaPaDiscreteStatsParameter class

~FaPaDiscreteStatsParameter(void)

This member function is a default destructor for the FaPaDiscreteStatsParameter class

Public Functions

Boolean **CheckIfStateChanged**(void)

This member function checks if the state of a variable is changed from the previous state

FaPaDiscreteStatsParameter& **GetResults**(void)

This member function is called by parameter group when the day interval is up.

void **RestoreGuts**(void)

This member function gets the different states, their duration and count from the dataset file

void **SaveGuts**(void)

This member function saves different parameter states, their total duration, total count and number of state changes on a dataset file.

FaPaDiscreteStatsParameter& **UpdateParameter**(void)

This member function is called from the parameter group to update the parameter with the new value

void **UpdateSameState**(void)

This member function updates the discrete parameter with the same state as that of a previous instance of the same parameter

void **UpdateStateChange**(void)

This member function updates the information about the previous states, upon receiving the new state. It also resets the attributes for a new state

void **UpdateStateChange**(void)

This member function updates the information about the previous states, upon receiving the new state. It also resets the attributes for a new state

Private Data

FaPaDiscreteParameterData* **myDiscreteData**

This member variable provides the pointer to an object FaPaDiscreteParameter which contains the statistics for a day interval

EctInt **myLastState**

This member variable stores the previous state of a parameter for comparison with that of a current value

EctInt **myStateChangeCounter**

This member variable provides the total number of times that the state has been changed in a day

container* **myStateList**

This member variable provides a list of different states associated with a given discrete parameter

FaPaLimitsParamGroup

class **FaPaLimitsParamGroup**

Base Classes

public **FaPaParameterGroup**

Public Construction

void**FaPaLimitsParamGroup**(FaReRequestParameter*)

contructor The parameter group must instantiate all FaPaLimitsParameters needed by the group using the list of RequestParameters. The FaPaLimitsParameters will point to the appropriate UserParameters using the address supplied in the by the RequestParameter

void**~FaPaLimitsParamGroup**(void)

destructor

Public Functions

const FaPaAnalysisParameter& **ProcessNewParameters**(const EctInt& NewlyArrivedParameterList)

ProcessNewParameters.

This is the function which updates the FaPaLimitsParameters. This will simply tell an FaPaLimitsParameter to use the new telemetry value to update itself and return a value based on the results of the update. Results will be generated when a limit violation changes, by either going back in limits or changing to a different violation.

FaPaLimitsParameter

class **FaPaLimitsParameter**

Base Classes

public **FaPaAnalysisParameter**

Public Types

class **FaPaLimitsParameter**

This class is derived from the FaPaAnalysisParameter class. It contains the attributes and operations necessary to capture the limit violations for the telemetry parameter and logs them on to a central dataset.

Base Classes

public **FaPaAnalysisParameter**

Public Construction

FaPaLimitsParameter(const FaReRequestParameter& reqParameterPtr)

This member function is a default constructor for the FaPaLimitsParameter

~FaPaLimitsParameter(void)

This member function is a default destructor for the FaPaLimitsParameter

Public Functions

EctBoolean **CheckIfViolationChanged**(void)

This member function checks if parameter value shows any limits violation and if it is different from the previous limit violation

void **ComputeDuration**(void)

This member function computes the duration of a limits violation from the start and end time of the limits violation

`void` **`Reset`**`()`

This member function resets limits related attributes like the last violation type, last violation time, previous violation duration etc to get ready for the next limit violation data

`void` **`RestoreGuts`**`()`

This member function reads in the limits violation information from the dataset file.

`void` **`SaveGuts`**`()`

This member function writes the limits violation information on the dataset file.

`FaPaLimitsParameter&` **`UpdateParameter`**`()`

UpdateParameter()

This member function updates the limits violation parameter when the current limits value has remained the same from the previous one. But when the limits violation changes, it returns the updated parameter to the calling object so that the calling object can write the Limits Violation information to the output stream

Private Functions

`enum(high, low, delta, normal)`

This member variable holds the Limits Violation type

`enum(high, low, delta, normal)`

This member variable stores the Limits Violation state from the current parameter

Private Data

`EctTime` **`myLastSCTime`**

This member variable stores the S/C time from the current parameter

`FaPaLimitsParameterData*` **`myLimitsData`**

This member variable points to an FaPaLimitsParameterData object which performs the read and write to a stream I/O. It also provides limits data into ASCII format for reports and carry-out data

`EctFloat` **`myLimitsViolationDuration`**

This member variable contains the total Limits Violation duration for a single contiguous Limits Violation period

`EctTime` **`myLimitsViolationSCTime`**

This member variable contains the S/c time from the parameter, when the Limits Violation was detected

FaPaLimitsParameterData

`class` **`FaPaLimitsParameterData`**

Public Types

`class` **`FaPaLimitsParameterData`**

This class stores the Limits Violation data to write to an output stream. It provides various functions to get Limits Violation components in ASCII or binary format.

Public Construction

`FaPaLimitsParameterData``(void)`

This member function is a default constructor for the FaPaLimitsParameter

`~FaPaLimitsParameterData``(void)`

This member function is a default constructor for the FaPaLimitsParameter

Public Functions

`String` **`getASCIIParameterID`**`(void)`

This member function returns myParameterID in character format

`String` **`getASCIIViolationDuration`**`(void)`

This member function returns myViolationDuration in character format

`String` **`getASCIIViolationSCTime`**`(void)`

This member function returns myViolationSCTime in character format

`String` **`getASCIIViolationType`**`(void)`

This member function returns myViolationType in character format

`EcTInt` **`getParameterID`**`(void)`

This member function returns myParameterID in binary format

`EcTFloat` **`getViolationDuration`**`(void)`

This member function returns myViolationDuration in binary format

`EcTTime` **`getViolationSCTime`**`(void)`

This member function returns myViolationSCTime in EcTTime format

`EcTInt` **`getViolationType`**`(void)`

This member function returns myViolationType in binary format

`String` **`makeCarryOut`**`(void)`

This member function provides the member data into a contiguous string which could be written to a carry-out dataset

`void` **`restoreguts`**`(void)`

This member function reads all attributes of this class from an input stream

`void` **`saveguts`**`(void)`

This member function writes all the attributes of this class on an output stream

`void` **`setParameterID`**`(const EcTInt ParameterID)`

description of operation

`void` **`setViolationDuration`**`(const EcTFloat ViolationDuration)`

This member function sets the myViolationDuration to ViolationDuration

`void` **`setViolationSCTime`**`(const EcTTime SCTime)`

This member function sets the myViolationSCTime to SCTime

`void` **`setViolationType`**`(const EcTInt ViolationType)`

This member function sets the myViolationType to ViolationType

Private Functions

`enum(high, low, delta, normal)`

This member variable stores the Limits Violation Type information. Limits Violation type could be high, low, delta or normal

Private Data

`EcTInt` **`myParameterID`**

This member variable stores parameter identification number

`EcTFloat` **`myViolationDuration`**

This member variable stores the Limits Violation duration for the parameter

`EcTTime` **`myViolationSCTime`**

This member variable stores the Limits violation spacecraft time when the violation started

FaPaLimitsParameterData

class **FaPaLimitsParameterData**

Public Types

class **FaPaLimitsParameterData**

This class stores the Limits Violation data to write to an output stream. It provides various functions to get Limits Violation components in ASCII or binary format.

Public Construction

FaPaLimitsParameterData(void)

This member function is a default constructor for the FaPaLimitsParameter

~FaPaLimitsParameterData(void)

This member function is a default constructor for the FaPaLimitsParameter

Public Functions

String **getASCIIParameterID**(void)

This member function returns myParameterID in character format

String **getASCIIViolationDuration**(void)

This member function returns myViolationDuration in character format

String **getASCIIViolationSCTime**(void)

This member function returns myViolationSCTime in character format

String **getASCIIViolationType**(void)

This member function returns myViolationType in character format

EcTInt **getParameterID**(void)

This member function returns myParameterID in binary format

EcTFloat **getViolationDuration**(void)

This member function returns myViolationDuration in binary format

EcTTime **getViolationSCTime**(void)

This member function returns myViolationSCTime in EcTTime format

EcTInt **getViolationType**(void)

This member function returns myViolationType in binary format

String **makeCarryOut**(void)

This member function provides the member data into a contiguous string which could be written to a carry-out dataset

void **restoreguts**(void)

This member function reads all attributes of this class from an input stream

void **saveguts**(void)

This member function writes all the attributes of this class on an output stream

void **setParameterID**(const EcTInt ParameterID)

description of operation

void **setViolationDuration**(const EcTFloat ViolationDuration)

This member function sets the myViolationDuration to ViolationDuration

void **setViolationSCTime**(const EcTTime SCTime)

This member function sets the myViolationSCTime to SCTime

`void` **`setViolationType`**`(const EcTInt ViolationType)`

This member function sets the myViolationType to ViolationType

Private Functions

`enum(high, low, delta, normal)`

This member variable stores the Limits Violation Type information. Limits Violation type could be high, low, delta or normal

Private Data

`EcTInt` **`myParameterID`**

This member variable stores parameter identification number

`EcTFloat` **`myViolationDuration`**

This member variable stores the Limits Violation duration for the parameter

`EcTTime` **`myViolationSCTime`**

This member variable stores the Limits violation spacecraft time when the violation started

FaPaOrbitStatsInterval

class **FaPaOrbitStatsInterval**

Base Classes

`public` **`FaPaStatsInterval`**

Public Functions

`void` **`LoadNewOrbit`**`(EctInt OrbitNumber)`

Private Data

`EctTime` **`myOrbitDayStartTime`**
`EctTime` **`myOrbitNightStartTime`**
`EctInt` **`myOrbitNumber`**

FaPaParameterGroup

class **FaPaParameterGroup**

This class is the base class for the parameter groups. A parameter group is a collection of parameters which perform some function and return zero or more parameters as results. Each parameter group points to a container of analysis parameters. The default action of the base class parameter group is to update all parameters which have updated in the most recent data packet. This will generally be all parameters received in a single packet. Other param groups may call algorithms which use parameter values, or update statistical values.

Public Construction

`FaPaParameterGroup``(FaReRequestParameter* RequestParameterList)`
`~FaPaParameterGroup``(void)`

contructor

```
The parameter group must instantiate all FaPaAnalysisParameters
needed by the group
using the list of parameter IDs sent to it. The
FaPaAnalysisParameters will point to the appropriate
UserParameters using the
address supplied in the by the RequestParameter
```

Public Functions

`virtual const FaPaAnalysisParameter&` **`ProcessNewParameters`**`(const EctInt&`

`NewlyArrivedParameterList)`

This is the function which updates the FaPaAnalysisParameters. This will simply cause the FaPaAnalysisParameter to use the new telemetry value to update itself and return a value based on the results of the update.

Private Data

`Container* myAnalysisParameterList`

myAnalysisParameters

This member is the container of FaPAAnalysisParameters the group needs to do it's function

`Container* myAnalysisResults`

this data member contains the results of the ProcessNewParameters member function

`EctTime myCurrentPacketTime`

This data member is the time of the current packet being processed.

`EctInt& myNewParamList`

This data member is a list of analysis parameter IDs which have updated in telemetry

FaPaStatsInterval

class FaPaStatsInterval

This class is used to check if the user specified time interval has expired.

Public Construction

`FaPaStatsInterval(void)`

This member function is a default constructor for the FaPaStatsInterval class

`~FaPaStatsInterval(void)`

This member function is a default destructor for the class FaPaStatsInterval

Public Functions

`Boolean CheckExpiration(const EctTime CurrentTime)`

This member function checks if the interval is up by using the current time of the parameter

`void ComputeEndTime(const EctInt Interval)`

This member function computes the end time of an interval using the interval provided by the user

`void Reset(EctTime CurrentTime)`

This member function resets the FaPaStatsInterval object with the end time of the previous interval

Private Data

`EctTime myEndTime`

This member variable gives the end time of an interval

`EctTime myStartTime`

This member variable gives the start time of an interval

FaPaStatsParameter

class FaPaStatsParameter

This class is derived from the FaPaAnalysisParameter. It is a base class for FaPaSystemsStatsParameter and FaPaUserStatsParameter class. The attributes and operations are used to calculate statistical values like Min, Max, Mean etc.

Base Classes

`public` **`FaPaAnalysisParameter`**

Public Construction

`FaPaStatsParameter``(void)`

This member function is a default constructor for the FaPaStatsParameter class

`~FaPaStatsParameter``(void)`

This member function is a default destructor for the FaPaStatsParameter class

Public Functions

`void` **`ComputeMean`**`(void)`

This member function computes Average value from the Total

`void` **`ComputeStdDev`**`(void)`

This member function computes the Standard deviation from the SumOfSquare value

`void` **`DetermineMin`**`(void)`

This member function determines Minimum out of two values

`void` **`DeterminwMax`**`(void)`

This member function determines Maximum out of two values

`void` **`IncrementSumOfSquare`**`(void)`

This member function increments SumOfSquare variable by the square value of current parameter

`void` **`IncrementTotal`**`(void)`

This member function adds current parameter value to the Total

`void` **`SetIntervalStartTime`**`(void)`

This member function assigns start time of the interval to myIntervalStartTime

`FaPaStatsParameter&` **`UpdateParameter`**`()`

UpdateParameter()

This member function is a virtual base class which updates newly arrived parameter

Private Data

`EctFloat` **`MyMean`**

This member variable gives the Mean value from a set of parameter
values

`EctFloat` **`MySCTimeAtMaximum`**

This member variable gives the S/C time for the highest valued parameter within the given interval

`EctFloat` **`MyTotal`**

This member variable provides the sum of all parameter values

`EctTime` **`myIntervalStartTime`**

This member variable gives the start time of an interval

`EctFloat` **`myMaximun`**

This member variable gives the highest value from the set of parameter values within an interval

`EctFloat` **`myMinimum`**

This member variable gives the lowest value from the set of parameter values within an interval

`EctInt` **myNumberOfSamples**

This member variable gives the count of total number of parameters to calculate Mean values

`EctFloat` **mySCTimeAtMinimum**

This member variable gives the S/c time for the lowest valued parameter within the given interval

`EctFloat` **myStdDev**

This member variable gives the standard deviation on the set of parameter values within a given interval

`EctFloat` **mySumOfSquare**

This member variable gives the total of all the square values of the parameters for a given time interval

FaPaSystemStatsParamGroup

class **FaPaSystemStatsParamGroup**

FaPaStatsParameterGroup

This class is a base class for system generated stats. This common piece of all these classes is an association with a FaPaStatsInteval which applies to all parameters in the group.

Base Classes

`public` **FaPaParameterGroup**

Public Construction

FaPaSystemStatsParamGroup`(void)`
~FaPaSystemStatsParamGroup`(void)`

Public Functions

`EctInt` **CheckInterval**`(void)`

This function will check if the interval has expired and reset the interval if it has expired. The return value is a true/false indicator of expiration

`void` **ProcessNewParameters**`(const EctInt& NewlyArrivedParameterList)`

This function does nothing, since the class is intended to be a virtual base class

Private Data

`FaPaStatsInterval*` **myInterval**

FaPaStatsInterval

This attribute is the timer used to determine when the statistical interval has expired. Possible intervals are daily and orbital.

FaPaSystemsStatsParameter

class **FaPaSystemsStatsParameter**

This class updates Systems Statistics parameter with the new data. When the interval is up for any analysis products (orbit statistics, daily statistics, monthly statistics, mission-to-date statistics etc.), this class writes the statistics to a corresponding dataset file.

Base Classes

`public` **FaPaStatsParam**

Public Construction

FaPaSystemsStatsParameter`(void)`

This member function is a default constructor for the FaPaSystemsStatsParameter class

~FaPaSystemsStatsParameter(void)

This member function is a default destructor for the FaPaSystemsStatsParameter class

Public Functions

FaPaSystemStatsParameter* **ComputeStatistics**(void)

This member function computes statistical values like Mean, Standard Deviation etc.

void **Reset**(void)

This member function initializes various attributes to zero

void **SaveGuts**(ostream&)

WriteToDataset

This member function writes to a dataset file using ostream

void **UpdateStatisticsWithParameterValues**(EctInt* ParameterIdList)

This member function updates statistical data with the new data.

FaPaUserStatsParamGroup

class **FaPaUserStatsParamGroup**

This is a group of user stats parameters. User stats params each have a unique interval, from 1 sec to 24 hours. results from user stats are generated every time a user specified interval expires.

Base Classes

public **FaPaStatsParamGroup**

Public Construction

FaPaUserStatsParamGroup(FaReUserStatsRequestParameter)
~FaPaUserStatsParamGroup(void)

Public Functions

const FaPaAnalysisParameter& **ProcessNewParameters**(const EctInt& NewlyArrivedParameterList)

This function will ask each of the parameters in

```
myAnalysisParameters if it's user specified
interval has expired,
and store the statistical information in myResults if
the interval has expired. It will then
return the address of myResults if results are present.
```

FaPaUserStatsParameter

class **FaPaUserStatsParameter**

This class is derived from the FaPaStatsParameter. It contains attributes and operations necessary to compute user defined statistics.

Base Classes

public **FaPaStatsParameter**

public **FaPaStatsParam**

Public Construction

FaPaUserStatsParameter(void)

This member function is a default constructor for the FaPaUserStatsParameter class

~FaPaUserStatsParameter(void)

This member function is a default destructor for the

```
FaPaUserStatsParameter class
```

Public Functions

Boolean **CheckIfIntervalIsUp**(const EctTime PacketTime)

This member function checks if the interval is up

FaPaUserStatsParameter* **ComputeMMM**(void)

This member function computes statistics when the user defined interval is up

void **Reset**(void)

This member function resets all the statistical counter to start with the next interval

void **UpdateStatisticsWithParamValues**(EctInt* ParameterIdList)

This member function updates the FaPaUserStatesParameter object with the new values from the Userparameter

FaPaStatsInterval* **UpdateStatsInterval**(void)

This memberfunction resets the FaPaStatsInterval with the new start and end time

Private Data

EctInt **myInterval**

This member variable holds span of seconds time between the intervals

EctTime **myStartTime**

This member variable holds the start time (S/c time)of the User defined statistics

FaPaStatsInterval* **myStatsInterval**

This member variable defines pointer to an object of type FaPaStatsInterval

FaPrAnalysisProduct

class **FaPrAnalysisProduct**

This is the base class for analysis products. This class will receive a list of parameter ID's from the OfflineDataManager and notify the parameter group of their arrival.

Public Construction

FaPrAnalysisProduct(FaReAnalysisRequest*)

FaPaAnalysisProduct constructor with request information.

~FaPrAnalysisProduct(void)

~FaPaAnalysis

```
This member function is the default destructor.
```

Public Functions

EctInt **Update**(ParameterList)

This member function receives a container of parameter ID's for all new parameters and invokes ProcessNewParameters for each ParameterGroup.

Private Data

Container* **myAnalysisResults**

This data member contains the results of the FaPaParameterGroup::ProcessNewParameters() class member function.

`String` **myFilename**

myFilename;

`This data member is used to identify the analysis product as specified`

by the analysis request.

`FaPaParameterGroup*` **myParameterGroupList**

This data member is a list of all parameter groups associated with this product.

`FaReAnalysisRequest*` **myRequest**

This data member points to the analysis request.

FaPrDailySystemStatsDataset

class **FaPrDailySystemStatsDataset**

This class uses Orbital statistics to build calendar day

`statistics.`

Base Classes

`public` **FaPrSystemStatsDataset**

Public Functions

`void` **ConvertToCarryout**`(void)`

This member function converts a daily system stats dataset to a carryout format.

`void` **CreateDataset**`(void)`

This function opens the dataset file to the point where

`new stats are inserted.`

`void` **GenerateReport**`(void)`

This member function generates an daily system stats report in human readable ASCII.

`void` **UpdateMissionToDateDataset**`(void)`

This member function updates the mission to date dataset.

`void` **WriteDataset**`(void)`

This member function invokes FaPaSystemStatsParameter::saveguts() for each parameter contained in myAnalysisResults.

Private Data

`String` **myFilename**

FaPrDataset

class **FaPrDataset**

This class will perform all function of the base class analysis product, as well as write results to a dataset file, convert the dataset to carryout format and generate a human readable ASCII report from a dataset.

Base Classes

`public` **FaPrAnalysisProduct**

Public Functions

`virtual EctInt` **ConvertToCarryout**`(void)`

This member function writes a analysis product to a file in carryout format.

`virtual EctInt` **`CreateDataset`**`(void)`

This member function opens a file and invokes a function call to CreateHeader().

`virtual EctInt` **`CreateHeader`**`(void)`

This member function writes a header for a product.

`virtual EctInt` **`GenerateReport`**`(void)`

This member function generates a report in human readable ASCII.

`virtual EctInt` **`WriteDataset`**`(void)`

This member function invokes FaPaAnalysisParameter::saveguts() for each parameter contained in myAnalysisResults.

Private Data

`String` **`myCreationTime`**

Creation time of the product.

`ostream` **`myOstream`**

output stream for dataset file

FaPrDiscreteStateDataset

`class` **`FaPrDiscreteStateDataset`**

This class writes discrete state stats to a dataset file, converts the dataset to carryout format and generate

`a human readable ASCII report from a dataset.`

Base Classes

`public` **`FaPrDataset`**

Public Functions

`virtual EctInt` **`ConvertToCarryout`**`(void)`

This member function writes a analysis product to a file in carryout format.

`virtual EctInt` **`CreateDataset`**`(void)`

This member function opens a file and invokes a function call to CreateHeader().

`virtual EctInt` **`CreateHeader`**`(void)`

This member function writes a header for a DiscreteStateDataset.

`virtual EctInt` **`GenerateReport`**`(void)`

This member function generates a report in human readable ASCII.

`virtual EctInt` **`WriteDataset`**`(void)`

This member function invokes FaPaDiscreteStateParameter::saveguts() for each parameter contained in myAnalysisResults.

Private Data

`String` **`myCreationTime`**

Creation time of the product.

FaPrLimitsDataset

`class` **`FaPrLimitsDataset`**

This class writes Limits parameters to a file, converts the dataset to carryout format

`and generate a human readable`

ASCII report from a dataset.

Base Classes

`public` **`FaPrDataset`**

Public Functions

`virtual EctInt` **`ConvertToCarryout`**`(void)`

This member function writes a analysis product to a file in carryout format.

`virtual EctInt` **`CreateDataset`**`(void)`

This member function opens a file and invokes a function call to CreateHeader().

`virtual EctInt` **`CreateHeader`**`(void)`

This member function writes a header for a product.

`virtual EctInt` **`GenerateReport`**`(void)`

This member function generates a report in human readable ASCII.

`virtual EctInt` **`WriteDataset`**`(void)`

This member function invokes FaPaLimitsParameter::saveguts() for each parameter contained in myAnalysisResults.

Private Data

`String` **`myCreationTime`**

Creation time of the product.

FaPrMonthlySystemStatsDataset

`class` **`FaPrMonthlySystemStatsDataset`**

Base Classes

`public` **`FaPrSystemStatsDataset`**

Public Functions

`void` **`ConvertToCarryout`**`(void)`

This member function converts a monthly system stats format to a carryout format.

`void` **`CreateDataset`**`(void)`

This member function is the control function for this class. It invokes function calls to the other member functions for this class.

`void` **`CreateMonthlyDataset`**`(void)`

This member function

`void` **`GenerateReport`**`(void)`

This member function generates a monthly system statistics report in human readable ASCII.

`void` **`WriteDataset`**`(void)`

This member function invokes FaPaSystemStatsParameter::saveguts() for each parameter contained in myAnalysisiResults.

Private Data

`String` **`myFilename`**

myFileName This data member identifies the analysis product as specified by an analysis request.

FaPrOrbitSystemStatsDataset

class **FaPrOrbitSystemStatsDataset**

This class generates orbit, orbit day, and orbit night datasets and reports.

Base Classes

public **FaPrSystemStatsDataset**

Public Functions

void **CreateDataset**()

ConvertToCarryout This member function converts a orbit, orbit day or orbit night dataset to a carryout format.

void **GenerateReport**(void)

This member function generates a orbit, orbit day, or orbit night statistics report in human readable ASCII.

void **WriteDataset**(void)

This member function invokes the FaPaSystemStatsParameter::saveguts for each parameter contained in myAnalysisResults.

Private Data

String **myFilename**

data members

```
myFilename
This data member identifies a product file as specified
by an analysis request.
```

FaPrReport

class **FaPrReport**

This class generates a human readable ASCII report.

Base Classes

public **FaPrAnalysisProduct**

Public Functions

virtual EctInt **CreateHeader**(void)

This member function writes a header for a product.

virtual EctInt **CreateHeader**(void)

CreateReport This member function opens a file and invokes a function call to CreateHeader().

virtual EctInt **WriteReport**(void)

This member function calls FaPrAnalysisParameter::GetASCII functions

```
for each parameter
```

FaPrSubsystemReport

class **FaPrSubsystemReport**

stp/omt class definition 1747713

Base Classes

public **FaPrReport**

FaPrSystemStatsDataset

class **FaPrSystemStatsDataset**

FaPrLimitsDataset

```
This  class  writes a dataset to a file,
converts the dataset to carryout format and generate a human readable
ASCII report from a dataset.
```

Base Classes

public **FaPrDataset**

Public Functions

virtual EctInt **ConvertToCarryout**(void)

This member function writes a analysis product to a file in carryout format.

virtual EctInt **CreateDataset**(void)

This member function opens a file and invokes a function call to CreateHeader().

virtual void **CreateHeader**(void)

This member function writes a header for a product.

virtual EctInt **GenerateReport**(void)

This member function generates a report in human readable ASCII.

virtual EctInt **WriteDataset**(void)

This member function invokes FaPaSystemStatsParameter::saveguts() for each parameter contained in myAnalysisResults.

Private Data

String **myCreationTime**

Creation time of the product.

FaReAnalysisRequest

class **FaReAnalysisRequest**

This is an interface class used by an User Interface Subsystem to generate the Analysis request for the Analysis subsystem. The Analysis request is used to generate standard telemetry datasets, and/or user defined statistics dataset, and/or algorithms produced dataset or to generate the Systems Generated Statistics.

Public Construction

FaReAnalysisRequest(void)

This member function is a default constructor for the FaReAnalysisRequest class

~FaReAnalysisRequest(void)

This member function is a default destructor for the FaReAnalysisRequest class

Public Functions

void **AddStatisticsParameter**(const EcTInt ParameterID, const EcTInt Interval)

This member function adds the statistic parameter in the Analysis Request

`void` **`AddTLMParameter`**`(const EcTInt ParameterID, const EcTInt SampleRate)`

This member function adds one telemetry parameter and associated samplerate in the Analysis Request

`EcTInt` **`DeleteALGParameter`**`(const EcTInt ParameterID)`

This member function deletes the specified parameter from the Analysis Request. The parameterType indicates if the parameter is an input parameter or an output parameter or an key parameter

`void` **`DeleteStatisticsParameter`**`(const EcTInt ParameterID)`

This member function deletes the statistics parameter and associated interval from the analysis request

`void` **`DeleteTLMParameter`**`(const EcTInt ParameterId)`

This member function deletes the specified parameter and associated samplerate from the analysis request

`void` **`GetALGParameter`**`(EcTEnum Index, EcTInt* ParameterID, String* SymbolName, EcTEnum* SymbolDataType, EcTEnum* EuConversion, EcTEnum ParameterType)`

This member function provides the Algorithm parameter and the position within the array of parameters is indicated by the Index

`String` **`GetAlgorithmName`**`(const EcTnum Index)`

This member function provides the name of the Algorithm from the Analysis Request

`String` **`GetHostName`**`(void)`

This member function provides Host Name from the Analysis Request

`String` **`GetInputDataset`**`(void)`

This member function gets the name of the input dataset from the Analysis Request

`EcTInt` **`GetOverrideDBID`**`(void)`

This member function provides the OverrideDBID from the Analysis Request

`EcTInt` **`GetOverrideQualityFlag`**`(void)`

This member function provides the value of the OverrideQualityFlag from the Analysis Request

`void` **`GetParameterFromList`**`(EcTInt* ParameterID, EcTInt* Address)`

GetParameterList

This member function is used to get the parameterID and address from the list. This member function is used by the Analysis Subsystem and not used by the UserInterface Subsystem

`EcTInt` **`GetRequestID`**`(void)`

This member function gets RequestID value from the Analysis Request

`String` **`GetSpaceCraftID`**`(void)`

This member function provides SpacecraftID from the Analysis Request

`EcTTime` **`GetStartTime`**`(void)`

This member function gets the Star time (of the Telemetry data) value from the Analysis Request

`void` **`GetStatisticsParameter`**`(const EcTEnum Index, EcTInt* ParamaterID, EcTInt* Interval)`

This member function provides the statistics parameter and associated interval from the Analysis Request

`EcTTime` **`GetStopTime`**`(void)`

This member function provides stop time of the Telemetry data from Analysis Request

`EcTInt` **`GetSystemsStatisticsFlag`**`(void)`

This member function gets the value of a Systems Statistics Flag from the Analysis Request

`EcTInt` **`GetTLMOutput`**`(String* DatasetName, EcTInt* OutputFormat)`

This member function extracts DatasetName and Format of the data from Analysis Request

`EcTInt` **`GetTLMParameter`**`(const EcTEnum Index, EcTInt* ParameterID, EcTInt* SampleRate)`

This member function gets the Telemetry Parameter from the analysis request. The position of the parameter is indicated

```
by the index which can have following values: First, Last,
or Next
```

`EcTInt` **`GetTimer`**`(String& AlgorithmName)`

This member function provides the Timer value associated with the AlgorithmName, from the Analysis Request

`EcTInt` **`GetUserID`**`(void)`

This member function gets UserID value from the Analysis Request

`void` **`LoadFromFile`**`(String& Filename)`

This member function uses data from the file to build the Analysis Request

`void` **`SaveToFile`**`(String& Filename, OverwriteFlag)`

This member function saves the Analysis Request in a file.

`void` **`SetALGParameter`**`(const EcTInt ParameterID, const String& SymbolName, const EcTEnum SymbolDataType, const EcTEnum EuConversion, const EcTEnum ParameterType)`

This member function sets the Algorithm parameter in the Analysis Request. The Algorithm parameter could be an Input Parameter, or output parameter, or a key parameter

`void` **`SetAlgorithmName`**`(String& AlgorithmName)`

This member function sets an Algorithm Name in the Analysis Request

`void` **`SetHostName`**`(const String& HostName)`

This member function sets Host Name (from where the request is originated) in the Analysis Request

`void` **`SetInputDataset`**`(String& DatasetName)`

This member function sets the dataset name in the request. When the dataset is used as an input, the dataset is used as a source of telemetry data instead of replay data from the archive

`void` **`SetOverrideDBID`**`(const EcTInt DBID)`

This member function sets the OverrideDBID flag in the Analysis Request. When overrideDBID flag is set, the request overrides the DBID which was used by the Telemetry

`void` **`SetOverrideQualityFlag`**`(const EcTEnum)`

This member function sets the Override Quality Flag within the Analysis Request. When the override quality flag is set, it indicates that the bad quality telemetry is allowed in the analysis of data

`void` **`SetParameterIntoList`**`(EcTInt ParameterID, EcTInt Address)`

SetParameterList

This member function is used to set the parameterID and an associated address in a list. This function is used by the Analysis Subsystem only.

`void` **`SetRequestID`**`(EcTInt RequestID)`

This member function sets UserID value in the Analysis Request

`void` **`SetSpacecraftID`**`(const String& SPID)`

This member function sets spacecraftID value in the Analysis Request

`void` **`SetStartTime`**`(const EcTTime StartTime)`

This member function sets the StartTime value in the Analysis Request

`void` **`SetStopTime`**`(const EcTTime)`

This member function sets the stop time of the Telemetry data in the Analysis Request

`void` **`SetSystemsStatisticsFlag`**`(const EcTInt SystemStatisticFlag)`

This member function sets the SystemsStatistics flag in the Analysis Request. This flag indicates if the request is for the Systems Generated statistics

`void` **`SetTLMOutput`**`(String& DatasetName, const EcTInt OutputFormat)`

This member function sets DatasetName and OutputFormat of the dataset in Analysis request

`void` **`SetTimer`**`(const EcTInt Timer)`

This member function sets the Timer (which fires off the Algorithm) value in the Analysis Request

`void` **`SetUserID`**`(EcTInt UserID)`

Private Data

`String` **`myHost`**

This member variable provides the host name from where the request is originated

`String` **`myInputDataset`**

This member variable provides the Input dataset name to get the Analysis Request data from a file

`EcTInt` **`myOverrideDBID`**

This member variable is a database override flag. User can specify override flag to override normal database.

`EcTInt` **`myOverrideQualityFlag`**

This member variable is a Data quality override flag. User can specify override flag to override bad quality data inclusion

`FaPaParamIDAddress*` **`myParameterList`**

This member variable provides the list of parameters and their addresses as they are instantiated by the Offline DataManager. This information is used by the Analysis Products in the Analysis Subsystem.

`EcTInt` **`myRequestID`**

This member variable provides the unique identification of the Analysis Request

`String` **`mySpacecraftID`**

This member function provides the identification of the spacecraft like AM-1

`EcTTime` **`myStartTime`**

This member variable provides the start time of the collection of telemetry data

`FaReRequestStatistics*` **`myStatisticsData`**

This member variable is a pointer to a Statistics data. The statistics data is composed of datasetname and a list of telemetry parameters with the user defined interval for the statistical calculations

`EcTTime` **`myStopTime`**

This member variable provides the stop time of the collection of telemetry data

`FaReRequestTLM*` **`myTLMData`**

This member variable is a pointer to a Telemetry data specified in the request. The Telemetry data corresponds to Telemetry parameters and Algorithm data

`EcTInt` **`myUserID`**

This member variable provides the identification of the user

FaReReqAlgorithmParameter

`class` **`FaReReqAlgorithmParameter`**

This class defines the attributes and operations necessary for the parameters associated with the algorithm. This class is derived from FaReRequestParameter

Base Classes

`public` **`FaReRequestParameter`**

Public Construction

`FaReReqAlgorithmParameter``(void)`

This member function is a default constructor for the class

`~FaReReqAlgorithmParameter``(void)`

This member function is a default destructor for the class

Public Functions

`EcTInt` **`GetEuConversion`**`(void)`

This member function provides the value of myEuConversion to the caller.

`EcTInt` **`GetSymbolDataType`**`(void)`

This member function provides the Datatype of the Symbol to the caller.

`String` **`GetSymbolName`**`(void)`

This member function provides the value of mySymbolName variable to the caller. The symbolic name has one to one association with the parameterID

`void` **`SetEuConversion`**`(EcTInt ConversionFlag)`

This member function assigns the ConversionFlag(raw or EU) to myEuConversion variable

`void` **`SetSymbolDataType`**`(EcTInt DataType)`

This member function assigns the DataType (real or int) to mySymbolType variable

`void` **`SetSymbolName`**`(String Name)`

This member function associates symbolic name with the parameterID

Private Data

`String` **`mySymbolName`**

This member variable stores the symbolic name for myParameterID in the Analysis request

FaReReqStatisticsParameter

`class` **`FaReReqStatisticsParameter`**

This class defines attributes and operations needed by a statistics parameter for the User defined statistics. This class is derived from FaReRequestParameter class

Base Classes

`public` **`FaReRequestParameter`**

Public Construction

`FaReReqStatisticsParameter``(void)`

This member function is a default constructor for the class

`~FaReReqStatisticsParameter``(void)`

This member function is a default destructor for the class

Public Functions

`EcTInt` **`GetInterval`**`(void)`

This member function retrieves the interval value from myInterval variable of the class

`void` **`SetInterval`**`(EcTInt Interval)`

This member function assigns the interval value to myInterval variable of the class

Private Data

`EcTInt` **`myInterval`**

This member variable stores the User defined interval for each parameter. The Mean, Max, Min, etc. values are calculated based on this interval for the current parameter.

FaReReqTLMParameter

`class` **`FaReReqTLMParameter`**

This class defines the attributes and methods needed for the telemetry parameters in the analysis request.

Base Classes

`public` **`FaReRequestParameter`**

Public Construction

`FaReReqTLMParameter``(void)`

This member function is a default constructor for the class

`~FaReReqTLMParameter``(void)`

This is a default destructor for the class

Public Functions

`EcTInt` **`GetSampleRate`**`(void)`

This member function provides the sample rate of a parameter from an Analysis request

`void` **`SetSampleRate`**`(EcTInt SampleRate)`

This member function sets the sample rate (how often the parameter is collected) for the parameter in the Analysis request

Private Data

`EcTInt` **`mySampleRate`**

This member variable stores the sampling information for a telemetry parameter within an Analysis request

FaReRequestAlgorithm

`class` **`FaReRequestAlgorithm`**

This class defines the attributes and methods necessary for Algorithms which are used in Analysis Request. It allows user to specify Algorithm name, Input parameters, Output parameters and Key parameters used in the algorithm

Public Construction

`FaReRequestAlgorithm``(void)`

This member function is a default constructor for the FaReRequestAlgorithm class

`~FaReRequestAlgorithm``(void)`

This member function is a default destructor for the FaReRequestAlgorithm class

Public Functions

`void` **`AddParameter`**`(EcTInt ParamID, String Name, EcTInt Type, EcTInt ConversionFlag, EcTInt ParameterType)`

AddParameters

This member function adds algorithm parameter either in an Input List or OutputParameter List or Key parameter List.

The Type of parameter is indicated by the ParameterType argument.

`EcTInt` **`GetConversionFlag`**`(void)`

This member function retrieves the conversion flag information from the algorithm parameter

`FaReReqAlgorithmParameter*` **`GetParameter`**`(const EcTEnum Index, String& AlgorithmName, const EcTInt ParameterType)`

This member function retrieves either an Input or an Output or a key parameter from the list of parameters associated with the algorithm. The parameter type is indicated by ParameterType

`String` **`GetSymbolName`**`(void)`

This member function retrieves the Symbolic name of a algorithm parameter

`EcTInt` **`GetSymbolType`**`(void)`

This member function retrieves the Symbolic type of a algorithm parameter

`EcTInt` **`GetTimer`**`(void)`

This member function gets the timer information from the FaReReqAlgorithm class

`void` **`SetAlgorithmName`**`(String Name)`

This member function assigns the algorithm name to myAlgorithmName variable

`void` **`SetTimer`**`(const EcTInt TimeInterval)`

This member function assigns the Time Interval value to the myTimer attribute of FaReRequestAlgorithm class

Private Data

`String` **`myAlgorithmName`**

This member variable stores the name of a algorithm in the FaReReqAlgorithm class

`FaReReqAlgorithmParameter` **`myInputParameters`**

This member variable defines the Input parameters for FaReReqAlgorithm Class

`FaReReqAlgorithmParameter` **`myKeyParameters`**

This member variable defines the key parameters for FaReReqAlgorithm Class

`FaReReqAlgorithmParameter` **`myOutputParameters`**

This member variable defines the Output parameters for FaReReqAlgorithm Class

`EcTInt` **`myTimer`**

This member variable specifies the Time Interval for the algorithm to start

FaReRequestParameter

`class` **`FaReRequestParameter`**

This class defines the attributes and methods needed for the telemetry parameter in the FaReAnalysisRequest. This is the base class for the request parameters. The parameters derived from this class are: FaReReqTLMParameter, FaReReqStatisticsParameter, and FaReReqAlgorithmParameter

Public Construction

`FaReRequestParameter``(void)`

This member function is a default constructor for the class

`~FaReRequestParameter``(void)`

This member function is a default destructor for the class

Public Functions

`void` **`GetParameterID`**`(EcTInt ParameterID)`

This member function provides ParameterID from myParameterID attribute of the class

`void` **`SetParameterID`**`(EcTInt ParameterID)`

This member function assigns the parameterID to myParameterID attribute in the FaReRequestParameter class

Private Data

`EcTInt` **`myParameterID`**

This member variable stores the parameter ID number.

FaReRequestStatistics

`class` **`FaReRequestStatistics`**

This class defines the attributes and methods needed for the User Defined Statistics parameter in an Analysis Request. It is derived from FaReRequestParameter class

Public Construction

`FaReRequestStatistics``(void)`

This member function is a default constructor for the class

`~FaReRequestStatistics``(void)`

This member function is a default destructor for the class

Public Functions

`void` **`AddParameter`**`(EcTInt ParameterID, EcTInt Interval)`

This member function adds statistic parameter in the list of parameters to be processed by the Analysis request

`String` **`GetOutPutDataset`**`(void)`

This member function gets the dataset name from the myOutputDataset variable of the class

`FaReReqStatisticsParameter*` **`GetParameter`**`(void)`

This member function retrieves the statistic parameter from the list provided in the Analysis request

`void` **`SetOutputDataset`**`(String Name)`

This member function assigns the dataset name to myOutputDataset variable of the class. The results of User Defined statistics are stored in this dataset.

Private Data

`String` **`myOutputDataset`**

This member variable specifies the name of the dataset where the results of the statistics computation should be written.

`FaReReqStatisticsParameter*` **`myParameterList`**

This member variable is a list of statistics parameter which needs to be processed for an Analysis request

FaReRequestTLM

`class` **`FaReRequestTLM`**

This class defines the Telemetry parameters and Algorithms for an Analysis request submitted by an User.

Public Construction

`FaReRequestTLM``(void)`

This member function is a default constructor for the FaReRequestTLM class

~FaReRequestTLM(void)

This member function is a default destructor for the FaReRequestTLM class

Public Functions

void **AddALGParameter**(String& AlgorithmName, EcTInt ParamID, String& SymName, EctInt SymType, EcTInt ConvFlag, EcTEnum ParameterType)

This member function adds any of the input, output or key parameters for an algorithm given in the FaReRequestAlgorithm class Note: Input, Output or Key parameter is indicated by the

ParameterType

void **AddParameter**(EcTInt ParamID, EcTInt Rate)

This member function adds parameter and sample rate for each parameter in a parameterlist

void **DeleteALGParameter**(String& AlgorithName, const EcTInt ParamID, const EcTEnum ParameterType)

This member function deletes an entry of a parameter from the parameterlist associated with the Algorithm in the FaReRequestAlgorithm class Note: Input, Output or Key parameter is indicated by the

ParameterType

void **DeleteParameter**(EcTInt ParamID, EcTInt Rate)

This member function deletes an entry of a given parameter from the parameterlist, in the FaReRequestTLM class

void **GetALGParameter**(String& AlgorithmName, EcTInt ParamID, String& SymName, EcTInt SymType, EctInt ConvFlag, EcTEnum ParameterType)

This member function provides the parameter associated with the Algorithm in the FaReRequestAlgorithm class Note: Input, Output or Key parameter is indicated by the

ParameterType

EcTInt **GetAlgorithmName**(const EcTInt index)

This member function provides the Algorithm Name from the list of algorithms given in the FaReRequestAlgorithm class

void **GetInterval**(const EcTInt ParamID)

This member function provides an interval specified for a given parameter, from the list of parameters in the FaReRequestTLM class

String **GetOutPutDataset**(void)

This member function provides the name of OutputDataset in the FaReRequestTLM class

EctInt **GetOutPutFormat**(void)

This member function gets the output format of the dataset

FaReReqTLMParameter* **GetParameter**(const EctEnum Index)

This member function provides the ParameterID and Samplerate for a parameter stored in the parameterlist of the FaReRequestTLM class

EcTInt **GetTimer**(void)

This member function gets the Timer value from the FaReRequestAlgorithm class

EcTInt **SetAlgorithmName**(String& Name)

This member function assigns the Algorithm Name in the FaReRequestTLM class

void **SetOutPutFormat**(EctInt Format)

This member function sets the Output Format for an output dataset in the FaReRequestTLM

void **SetOutputDataset**(String& Name)

This member function sets the value of OutputDataset in the FaReRequestTLM class

`void` **`SetTimer`**`(EcTInt)`

This member function assigns a value to the Timer variable, in the FaReRequestAlgorithm class

Private Data

`Container*` **`myAlgorithmList`**

This member variable stores the list of Algorithms and associated parameters. The Algorithm list is defined by FaReRequestAlgorithm class

`String` **`myOutputDataset`**

This member variable stores the name of the output dataset. The results of analysis computations are written on this dataset. The dataset is used by UserInterface Subsystem for plotting or for generating reports or a spreadsheet.

`Enum` **`myOutputFormat`**

This member variable provides the format for the data written on the dataset. The output format could be Binary, ASCII or Carryout.

`Container*` **`myParameterList`**

This member variable stores the list of telemetry parameters used for generating the standard telemetry dataset. Each parameter is associated with the samplerate. The parameter type is defined by FaReReqTLMParameter class

3.3 Analysis Request Manager

The Analysis Request Manager controls all the replay requests coming from the Request Queue process. It runs as a permanent process on machines (in the EOC and IST network) which are available for the offline Analysis processing. It is designated with the local, global, IST or inactive state. Depending upon its state, the Request Queue decides which Replay requests to send to the Request Manager. The Replay request could be an Analysis request or it could be a dedicated Replay request. For the Analysis request, the Analysis Request Manager starts up an Analysis process which handles all the processing needed for the generation of the analysis products. Upon completion of the Analysis request, the Analysis process sends the completion status to the Analysis Request Manager. The Analysis Request Manager passes the completion status to the Request Queue process and terminates the Analysis process. For the dedicated Replay request, it requests the RMS for the Telemetry Service. The RMS starts up DECOM process and returns the DECOM process address to the Request Manager which in turn passes the DECOM address to the DMS Replay process. Upon receiving the Replay Request completion message from the Request Queue process, the Request manager sends Terminate message to RMS to terminate DECOM process.

The Analysis Request Manager keeps track of all the Replay requests which have been submitted to it. When the new request is received, the Analysis Request Manger check points the request to a file so that in the case of a system crash, the request could be run again. The intermediate status of all Analysis requests is provided by the Analysis processes in terms of their current packet times. When the requests get completed, they are removed from the Replay list and also from the file.

3.3.1 Analysis Request Manager Context

The Analysis Request Manager interacts with the following interfaces:

1. DMS (Request queue and DMS Replay)
2. RMS (Telemetry service)
3. Analysis Process (Analysis request)
4. FaRmStateChangeRequest

It receives telemetry Replay requests from the Request Queue process. The Request queue process controls the number of requests to send to a given Request Manager. Upon completion of the Analysis Replay request, the Request Manager sends the completion status to the Request queue process. If the user decides to abort the submitted Analysis request, the Terminate request is issued by the user and the Request Manager sends the termination message to the Analysis process to end the processing of a request. The Request Manager interacts with the RMS to get the DECOM service. It interacts with the multiple Analysis processes to process the Analysis requests. The FOT personnel could submit the StateChange request to change the Request Manager process's state.

-Refer to Analysis Request Manager Context Diagram Figure 3.3-1;

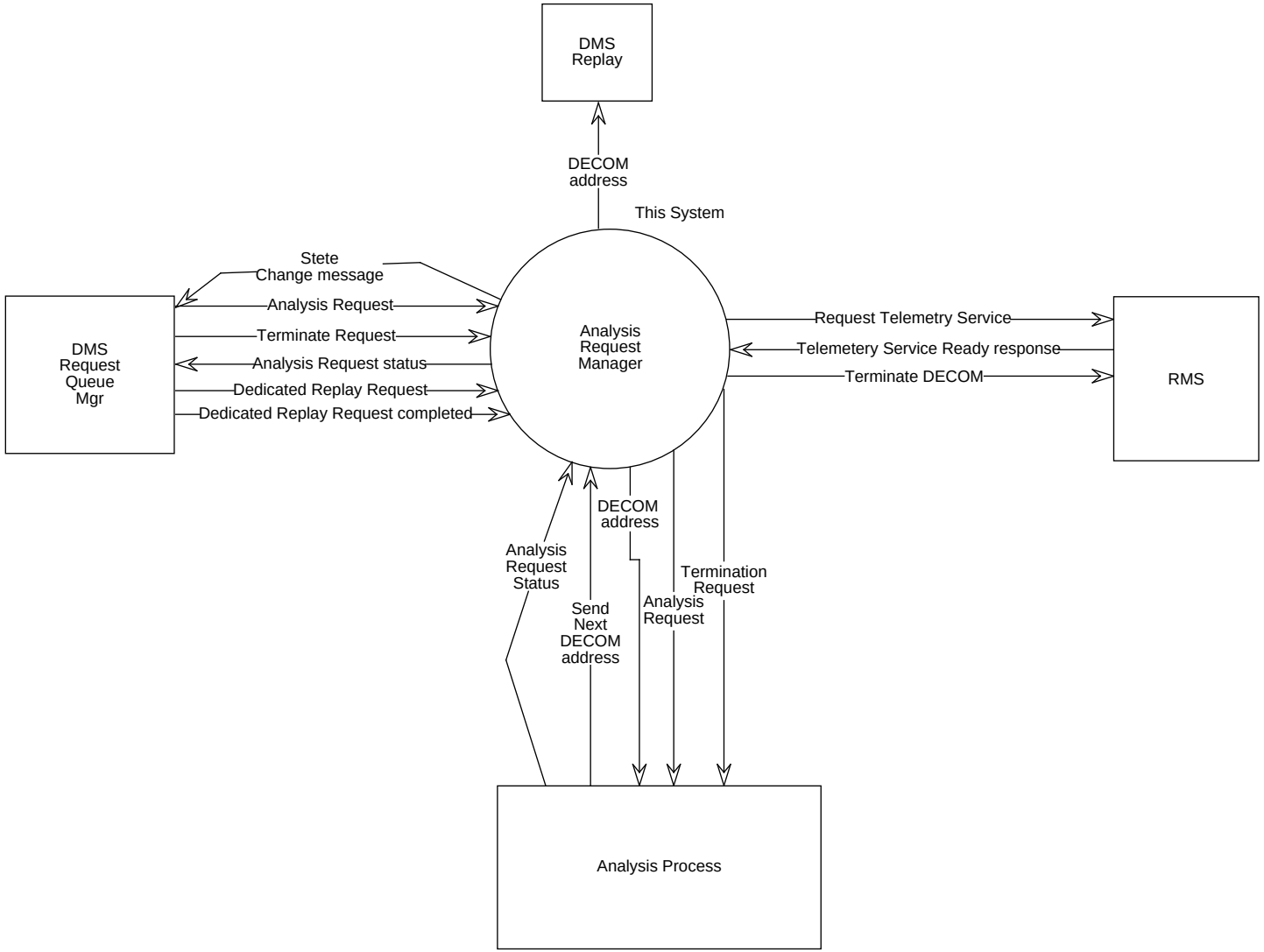

Figure 3.3-1. Analysis Request Manager Context

3.3.2 Analysis Request Manager Interface

***Table* 3.3.2. Analysis Request Manager Interface**

Interface Service	Interface Class	Interface Class Description	Service Provider	Service User	Frequency
Send TLM Replay Request	FaRpQueueToReqMgrProxy	Sends Telemetry Replay requests to RequestManager	ANA	DMS	Controlled by Request queue process
Send StateChange Request	FaRpScriptToReqMgrProxy	Sends StateChange Request	ANA	FOT	Not very frequent
Send TLM Service Request Receive TLM Service Ready Request	FrGrReplayRequestProxy	Sends request for Telemetry Service to RMS Receives response for Telemetry Service Ready from RMS	RMS	ANA	As many as an Analysis requests submitted
Send Analysis Request	FaRpReqMgrToAnaProxy	Sends Analysis request from Request Manager to an Analysis process	ANA	ANA	As many as Analysis requests

3.3.3 Analysis Request Manager Object Model

This model shows different classes and their interactions with each other to process the Telemetry Replay Requests. The FaRmRequestManager is the main class for the Request Manager process. It connects the Request Manager process with the DMS and RMS subsystems. It instantiates FaRmReplay objects and FaRmAnalysisProcess objects to accomplish Replay and request processing tasks. The FaReCompletionStatus represents the completion status of an analysis request. It is sent from the Analysis process to the Request Manager process and finally from the Request Manager to the Request Queue via FaRpQueueToReqMgrProxy. The FaRmAnalysisReqStatus class represents the intermediate status of the request and it provides the current packet time of the analysis request at a regular time interval. The FaRmAnalysisReqStatus is also passed on to the Request Queue via FaRpQueueToReqMgrProxy.

-Refer to Analysis Request Manager Object Model Figure 3.3-2;

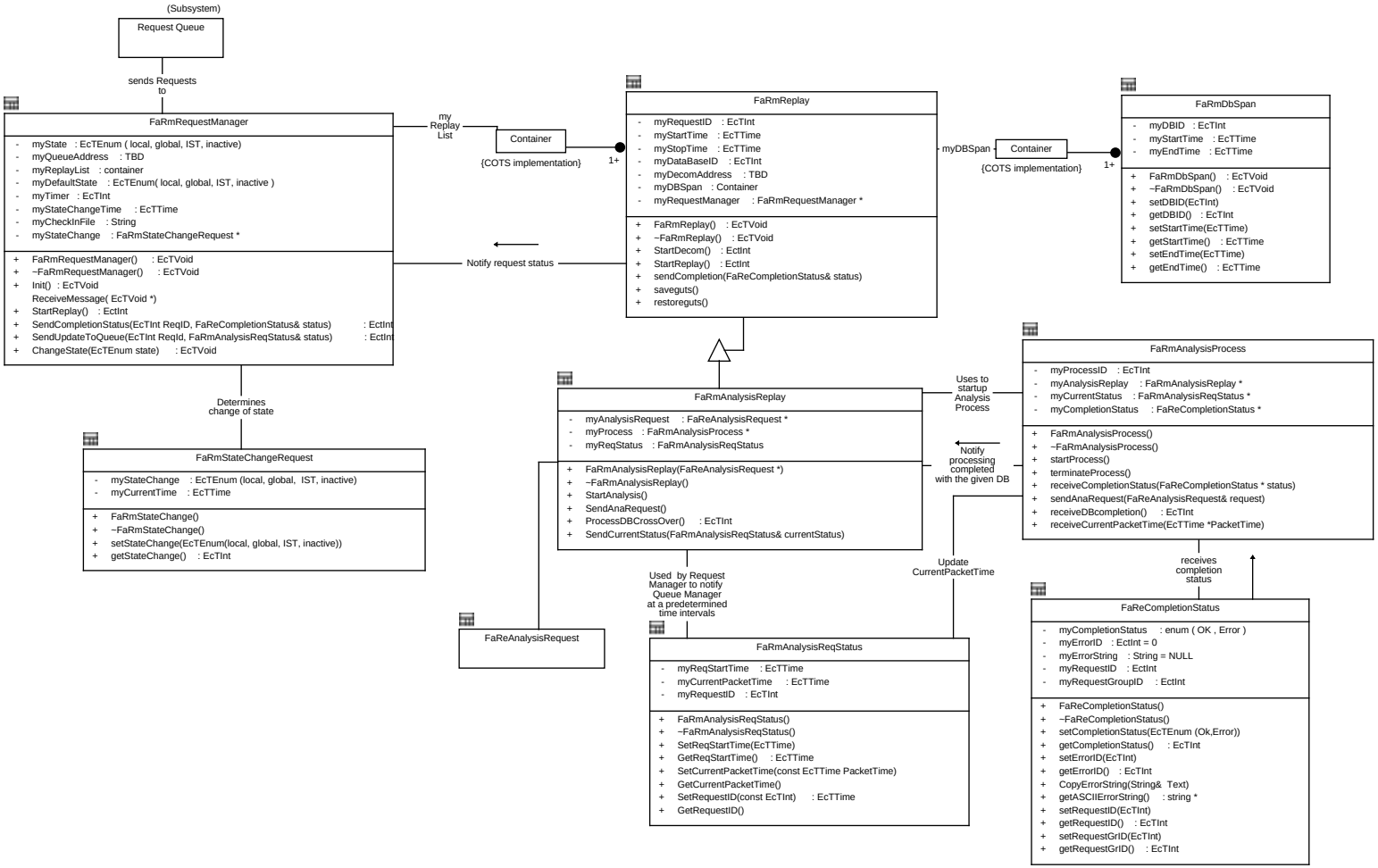

Figure 3.3-2. Analysis Request Manager Object Model

3.3.4 Analysis Request Manager Dynamic Model

This section details the Analysis Request Manager process.

3.3.4.1 Analysis Requests Processing Scenario Abstract

The event trace diagram describes the complete scenario about the Analysis requests processing done by the Analysis Request Manager process.

-Refer to Analysis Request Manager Event Trace Figure 3.3-3;

3.3.4.2 Analysis Requests Processing Summary Information

Interfaces:

FOS Data Management Subsystem

FOS Resource Management Subsystem

Stimulus:

The Analysis Request Manager will be running permanently on the predetermined workstations and it will start processing upon receiving either analysis or replay requests from the DMS Request Queue Mgr process. It handles multiple requests.

Desired Response:

The Analysis request manager sends the completion status response to the DMS Request Queue Mgr process and the Analysis product will be ready in the predetermined location for the further use by the User Interface subsystem.

Pre-Conditions: None

Post-Conditions: None

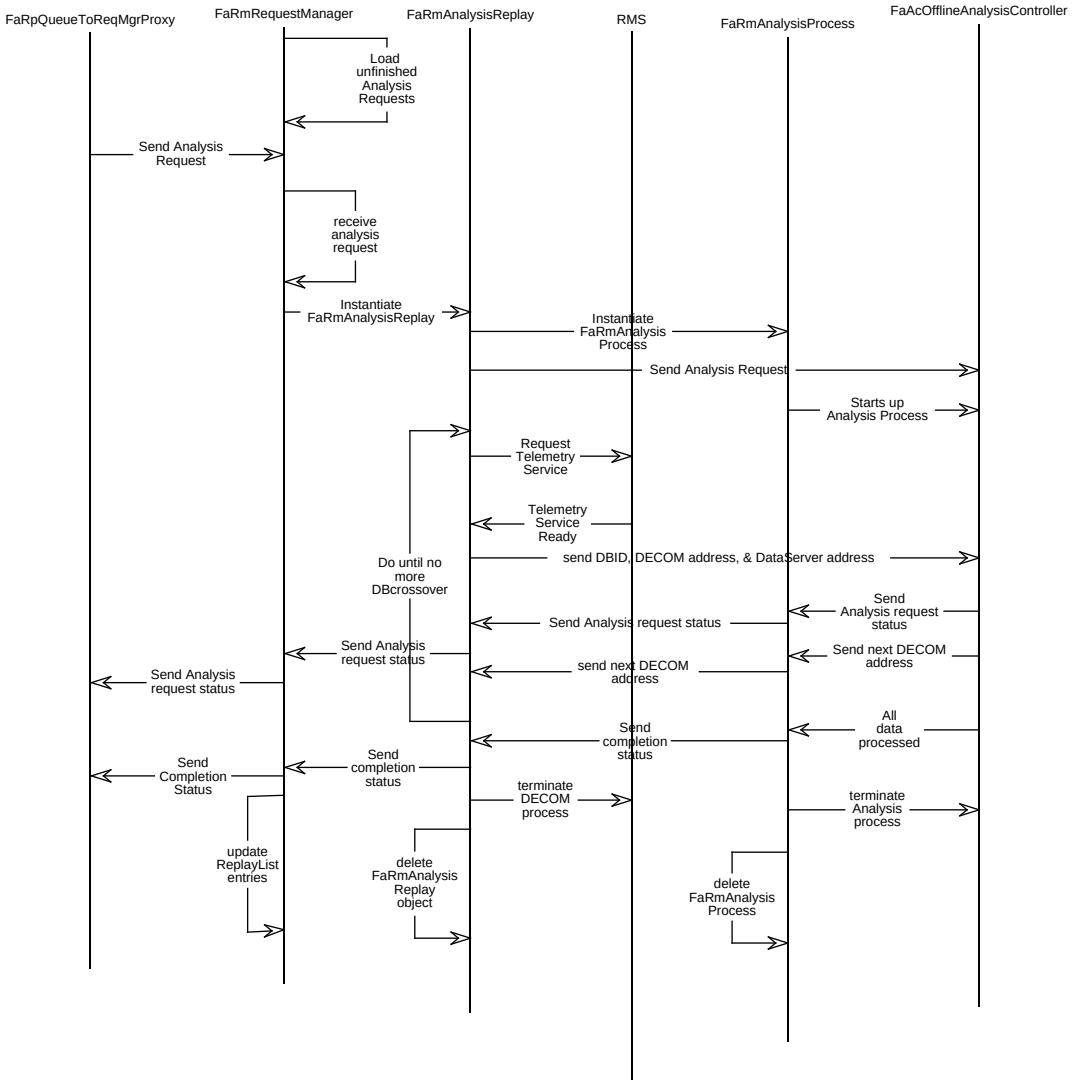

Figure 3.3-3. Analysis Request Manager Event Trace

3.3.4.3 Analysis Requests Processing Scenario Description

This model provides the event traces for the Analysis Request Manager process. The Telemetry Replay request is received by FaRmRequestManager via FaRpQueueToReqMgrProxy class. Upon receiving the request, if it is an Analysis request, the FaRmRequestManager instantiates the FaRmAnalysisReplay object, if it is a dedicated Replay request, it instantiates FaRmReplay object. The FaRmAnalysisReplay requests RMS to start up DECOM process for the telemetry decom. Upon receiving the response that the Telemetry Service is ready, it instantiates FaRmAnalysisProcess object to start up Analysis process. The FaRmAnalysisProcess starts up an Analysis process and establishes connection between the two processes. The FaRmAnalysisReplay object first determines the DBIDs required by the Analysis request and then sends the Analysis request to the Analysis process via IPC. It also sends DECOM and Dataserver process address to the Analysis process via IPC. When the Analysis process completes processing of the data, it sends the message to the FaRmAnalysisProcess object that it has completed processing with the given DECOM process. If the request spans more than one database, FaRmAnalysisReplay requests Telemetry service from the RMS and upon receiving one, it sends the DECOM address to the Analysis Process. when the Analysis request is completely processed, the Analysis process sends the completion status to the FaRmAmalysisProcess which in turn sends the message to the FaRmAnalysisReplay object. The FaRmAnalysisReplay object sends the completion status to the FaRmRequestManager which in turn sends the message to the Request queue process via IPC. As the message gets transferred from FaRmAnalysisProcess to FaRmRequestManager, the FaRmAnalysisProcess, FaRmAnalysisReplay objects are deleted and FaRmRequestManager updates Replay list.

3.3.5 Analysis Requests Processing Data Dictionary

FaReCompletionStatus

class **FaReCompletionStatus**

This class defines the attributes and methods used for saving the completion status of the Analysis request.

Public Construction

FaReCompletionStatus(void)

This member function is a default constructor for the FaReCompletionStatus

~FaReCompletionStatus(void)

This member function is a default destructor for the FaReCompletionStatus

Public Functions

void **CopyErrorString**(String& Text)
string* **getASCIIErrorString**(void)
EcTInt **getCompletionStatus**(void)
EcTInt **getErrorID**(void)
EcTInt **getRequestGrID**(void)
EcTInt **getRequestID**(void)
void **setCompletionStatus**()
void **setErrorID**(EcTInt)
void **setRequestGrID**(EcTInt)
void **setRequestID**(EcTInt)

Private Functions

`enum(OK, Error)`

This member variable stores the completion status of an Analysis request. It can have two values namely, OK or Error

Private Data

`EctInt` **`myErrorID`**

This member variable represents error mnemonic for an error condition encountered during the execution of an Analysis request

`String` **`myErrorString`**

This member variable stores the description of the error encountered during the execution of an Analysis request

`EctInt` **`myRequestGroupID`**

This member variable stores the request group id number for a request

`EctInt` **`myRequestID`**

This member variable stores the request id number for a request

FaRmAnalysisProcess

`class` **`FaRmAnalysisProcess`**

This class starts up and controls the Analysis process for each Analysis request. It is instantiated by the FaRmAnalysisReplay class. At a regular interval, it receives the current packet time from the Analysis process. It updates the FaRmAnalysisReqStatus object and passes it to the FaRmRequestManger. When the request is completed, it terminates the Analysis process, sends the completion status to the Request Manager and deletes itself.

Public Construction

`FaRmAnalysisProcess``(void)`

This member function is a default constructor for the class

`~FaRmAnalysisProcess``(void)`

This member function is a default destructor for the class

Public Functions

`void` **`receiveCompletionStatus`**`(FaReCompletionStatus*)`

This member function receives the request completion status from the Analysis process

`void` **`receiveCurrentPacketTime`**`(EcTTime& PacketTime)`

This member function receives the current packet time from the Analysis process at a regular time interval

`EcTInt` **`receiveDBcompletion`**`(void)`

This member function receives the partial completion message from an Analysis process when the Analysis Request time interval requires more than one database

`void` **`startProcess`**`(void)`

This member function starts up an Analysis process

`void` **`terminateProcess`**`(void)`

terminateProcess(void);

This member function terminates the Analysis process

Private Data

`FaRmAnalysisReplay* myAnalysisReplay`

This member variable holds the pointer to the FaRmAnalysisReplay object.

`FaRmAnalysisReqStatus* myCurrentStatus`

This member variable stores the pointer to the FaRmAnalysisReqStatus object which holds the current status of the request

`EcTInt myProcessID`

This member variable holds Analysis process ID. The analysis process is started by the FaRmAnalysisProcess object.

FaRmAnalysisReplay

class FaRmAnalysisReplay

This is class is derived from the FaRmReplay object. For each Analysis request, the FaRmRequestManger instantiates FaRmAnalysisReplay. It determines DBIDs based on the start and stop times of the request. It calls StartDecom to start DECOM process. It calls StartAnalysis function to instantiate FaRmAnalysisProcess object. It calls SendAnaRequest to send the Analysis request and DECOM address to the Analysis process via FaRmAnalysisProcess object. It sends the current status of the request to the Request Queue process via FaRmRequestManager object. Upon completion of the request, it sends completion message to the Request Queue process via FaRmRequestManager. It also deletes the instant of FaRmDbspan and deletes the connection between FaRmAnalysisReplay and FaRmRequestManager and with the RMS process. Finally it deletes itself.

Base Classes

`public FaRmReplay`

Public Construction

`FaRmAnalysisReplay(FaReAnalysisRequest*)`

This member function is a default constructor for the class

`~FaRmAnalysisReplay(void)`

This member function is a default destructor for the class

Public Functions

`EcTInt ProcessDBCrossOver(void)`

This member function handles DB crossover for the Analysis request. When data corresponding to one Database is processed, it calls startDecom function to startup another DECOM process which uses the new DB id table. The DECOM address is passed to the Analysis Process via FaRmAnalysisProcess object.

`void SendAnaRequest(void)`

This member function sends an Analysis request to the Analysis process via Inter process IPC.

`void SendCurrentStatus(FaRmAnalysisReqStatus& currentStatus)`

This member function sends the Current status of the request to the Request Queue via FaRmRequestManager object

`void StartAnalysis(void)`

This member function instantiates FaRmAnalysisProcess object to start an analysis process for a given Analysis Request. It also handles DB cross over for the request. It sends the current packet time to FaRmRequestManager.

Private Data

`FaReAnalysisRequest* myAnalysisRequest`

This member variable holds the values of the Analysis Request

`FaRmAnalysisProcess* myProcess`

This member variable holds the address of an FaRmAnalysisProcess object

FaRmAnalysisReqStatus

class **FaRmAnalysisReqStatus**

This class defines the attributes and operations needed for the Analysis request status

Public Construction

FaRmAnalysisReqStatus(void)

This member function is a default constructor for the class

~FaRmAnalysisReqStatus(void)

This member function is a default destructor for the class

Public Functions

void **GetCurrentPacketTime**(void)
void **SetCurrentPacketTime**(const EcTTime PacketTime)

This member function updates the Current Packet time within an object

EcTTime **GetReqStartTime**(void)

This member function gets the request start time from the object

void **GetRequestID**(void)
EcTTime **SetRequestID**(const EcTInt)

GetRequestID

void **SetReqStartTime**(EcTTime)

This member function sets the start time of the request

Private Data

EcTTime **myCurrentPacketTime**

This member variable stores the Current Packet time

EcTTime **myReqStartTime**

This member variable stores the Analysis request start time

EcTInt **myRequestID**

This member variable stores the Analysis request id

FaRmDbSpan

class **FaRmDbSpan**

This class defines the DBID entries which are used by DECOM process

Public Functions

EcTVoid **FaRmDbSpan**(const EcTInt DBID, const EcTTime startTime)
EcTInt **getDBID**(void)
EcTTime **getEndTime**(void)
EcTTime **getStartTime**(void)
void **setDBID**(EcTInt)
void **setEndTime**(EcTTime)
void **setStartTime**(EcTTime)

This member function is a default constructor of the class

Private Data

`EcTInt` **`myDBID`**

This member variable holds the DBID from the DataBase ID table. The database used during operation is identified by the DataBase ID.

`EcTTime` **`myEndTime`**

This member variable holds the end time when the database was used operationally.

`EcTTime` **`myStartTime`**

This member variable holds the start time when the database was used operationally.

FaRmReplay

`class` **`FaRmReplay`**

This is a base class for the Telemetry Replay. It determines DBIDs based on the start and stop times of the request. It requests RMS to start DECOM. The address of the DECOM process is sent to the DMS. Upon completion of the request, it sends completion message to the request queue via FaRmRequestManager. It also deletes the instant of FaRmDbspan and deletes the connection between FaRmReplay and FaRmRequestManager and with the RMS process. Finally it deletes itself.

Public Functions

`EcTVoid` **`FaRmReplay`**`(void)`

This member function is a default constructor for the class

`EctInt` **`StartDecom`**`(void)`

This member function sends request to RMS to start up Decom process

`EctInt` **`StartReplay`**`(void)`

This member function determines the DBIDs required by the request. It starts up DECOM process by calling StartDecom

`void` **`sendCompletion`**`(FaReCompletionStatus& status)`

This member function sends the request completion status object to Request queue via FaRmRequestManager

Private Data

`FaRmDbSpan*` **`myDBSpan`**

This member variable stores the pointer to a FaRmDbSpan object

`EcTInt` **`myDataBaseID`**

This member variable stores the DataBase ID specified in the request. If the DataBase ID is specified in the request, it overrides the operational DataBase

`TBD` **`myDecomAddress`**

This member variable stores the address of a DECOM process

`EcTInt` **`myRequestID`**

This member variable stores the Request Identifier for the Analysis request

`FaRmRequestManager*` **`myRequestManager`**

This member variable stores the pointer to its originating Request Manager

`EcTTime` **`myStartTime`**

This member variable stores the start time of the time span specified in the Request

`EcTTime` **`myStopTime`**

This member variable stores the end time of time span specified in the request

FaRmRequestManager

class **FaRmRequestManager**

This class represents an analysis process which provides the initial setup for the Analysis requests and Replay requests which come from the Request Queue process. For each Analysis request, it starts the Analysis process and controls that process. At a regular time interval each Analysis process updates the FaRmRequestManager with the current packet time. In turn, the FaReRequestManager updates Request Queue process with the status of each Analysis request. Upon receiving the terminate message from the Request Queue process, it terminates the Analysis request specified in the message. When the Analysis request is completed, the FaReRequestManager terminates the corresponding Analysis process and the sends the completion status to the Request Queue.

Public Functions

`EcTVoid` **`ChangeState`**`(EcTEnum state, EcTTime Time)`

Upon receiving the ChangeState request, FaRmRequestManager updates myState and myStateChangeTime member variables

`EcTVoid` **`FaRmRequestManager`**`(void)`

This member function is a default constructor for the FaReAnalysisRequest

`EcTVoid` **`Init`**`(void)`

This member function initializes the connections between the FaReRequestManager object and the external interfaces like DMS, RMS, etc.

`EcTInt` **`ReceiveMessage`**`(EcTVoid*)`

This member function receives different requests from the Request Queue process. The different requests includes: 1) Analysis request 2) Replay request and 3) Terminate request. Upon receiving Analysis or Replay request, it executes StartReplay function and then on the request execution continues.

`EctInt` **`SendStatus`**`(void)`

This member function sends the request completion status to the Request Queue process

`EctInt` **`SendUpdateToQueue`**`(void)`

When the timer expires, FaRmRequestManger updates the Request Queue process with the current status of all the request which are currently processed.

`EctInt` **`StartReplay`**`(void)`

This member function initiates the processing for the Replay.

Private Functions

`enum(local, global, IST, inactive)`

This member variable stores the default state

`enum(local, global, IST, inactive)`

This member variable stores the current state of the FaReRequestManager process.

Private Data

`String` **`myCheckInFile`**

This member variable provides the absolute pathname of a file used for storing the requests which could be executed later when the Request Manager process crashes and comes back again.

`TBD` **`myQueueAddress`**

This member variable stores the Request Queue process address

`container` **`myReplayList`**

This member variable contains the Analysis and Replay requests

`EcTTime` **`myStateChangeTime`**

This member variable stores the time when the FaRmRequestManager changes its state

`EcTInt` **`myTimer`**

This member variable stores the timer value. when the timer expires, FaReRequestManager sends status of all current requests to the Request Queue.

3.4 Spacecraft Clock Correlation Analysis

The Clock Correlation Analysis process provides the services of calculating discrepancies between the Spacecraft Clock and Coordinated Universal Time (UTC), generating reports about those discrepancies, and when necessary, generating Command Requests to adjust the Spacecraft clock. The process will calculate a Spacecraft Clock Error using two independent methods: the Return Data Delay (RDD) and the User Spacecraft Clock Calibration System (USCCS) algorithms.

Designated as the primary method used, the more accurate USCCS algorithm analyzes the recorded arrival and departure times of Pseudo random Noise(PN) Epochs traveling to and from the Spacecraft. This is accomplished through a Tracking and Data Relay Satellite System (TDRSS) Tracking Service. To calculate a Spacecraft Clock Error, the algorithm determines the UTC time of a PN Epoch's arrival at the Spacecraft and compares that time value with the Spacecraft Clock's reading of when the PN Epoch arrived at the Spacecraft. The RDD algorithm will be designated as the secondary method. Using time stamps in telemetry packets, this algorithm calculates the UTC time of when a telemetry packet was initiated on the Spacecraft and compares that time value with the Spacecraft Clock's reading of when a telemetry packet was initiated.

During a Real-Time contact, the White Sands Ground Terminal Upgrade (WSGTU) or the Secondary TDRSS Ground Terminal (STGT) transmits a Forward PN Epoch to the Spacecraft. Upon arrival at the Spacecraft, these Forward PN Epochs trigger the Spacecraft to transmit a Return PN Epoch back to the ground. The departure time of a Forward PN Epoch and the arrival time of a Return PN Epoch are both recorded at the WSGTU/STGT. These recorded PN Epoch times are then forwarded to the Network Controls Center (NCC). Also during a Real-Time contact, telemetry packages in the form of Coded Virtual Channel Data Units (CVCDUs) are received from the Spacecraft at the WSGTU/STGT. These CVCDUs are forwarded to the EOS Data and Operations System (EDOS).

EDOS reconstructs the telemetry packets from the CVCDUs and appends time stamped EDOS headers to the packet, thereby creating EDOS Data Units (EDUs) which are forwarded to the DECOM process. Upon termination of a Return Service, the NCC sends Operational Messages, OPM-62s, to the Real-Time Contact Management Subsystem (RCM). An OPM-62 contains equipment delay measurements associated with the telemetry packets; these measurements will be used in calculations during the next contact. After the completion of a TDRSS Tracking Service, the NCC also sends to RCM the recorded PN Epoch times along with PN Epoch related equipment delay measurements in another Operational Message, OPM-66.

-Refer to Figure 3.4-1 Clock Correlation Analysis High Level Data Flow

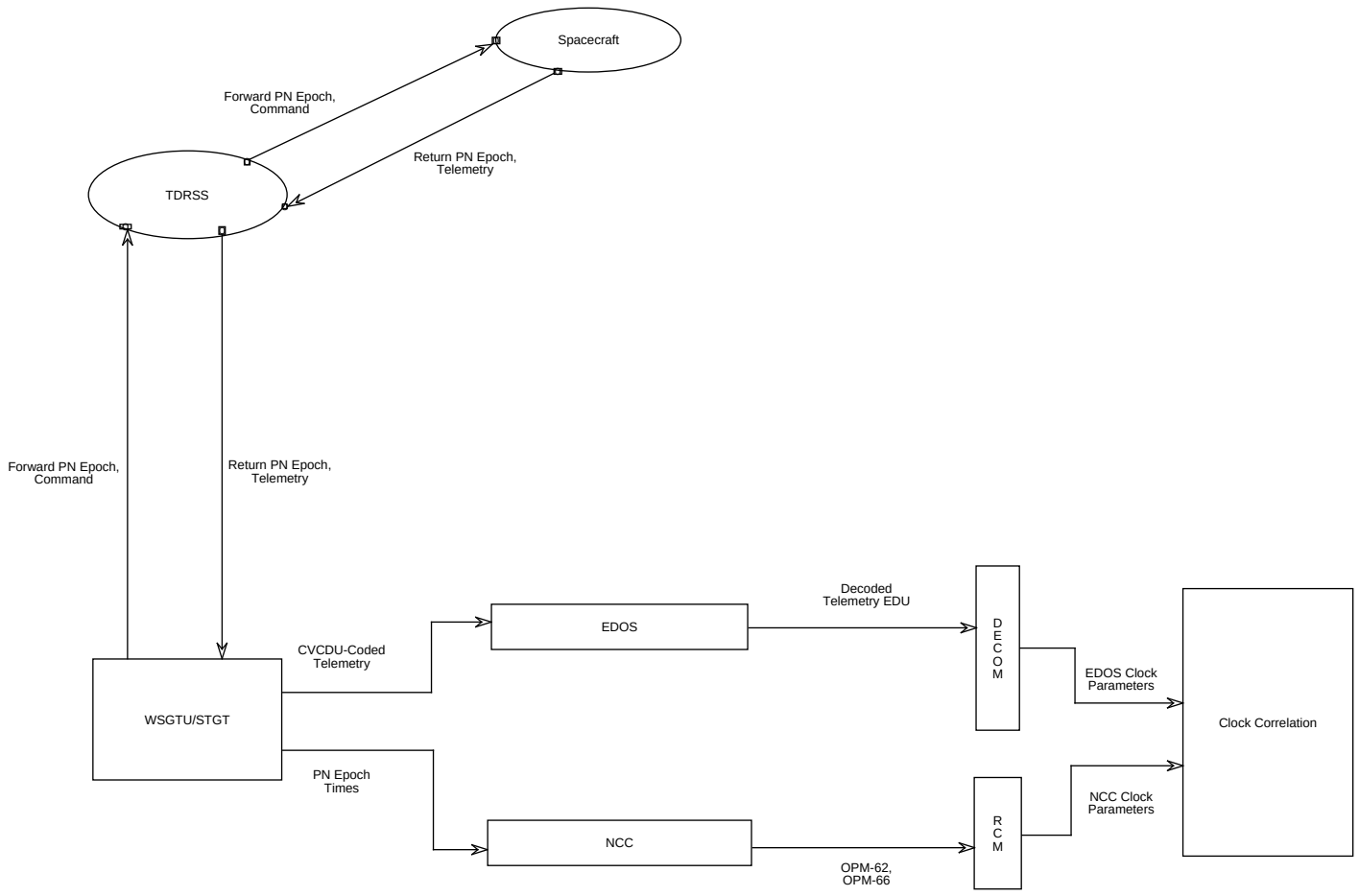

Figure 3.4-1. Clock Correlation Analysis High Level Data Flow

3.4.1 Clock Correlation Analysis Context

-Refer to Figure 3.4-2 Clock Correlation Analysis Context Diagram

The Decommutation (DECOM) process extracts specified clock parameters from the EDUs and forwards them to the Parameter Server. RCM extracts specified clock parameters from the OPM-62s and the OPM-66s and forwards them to the Parameter Server. The Clock Correlation Analysis process is notified by the Parameter Server when it has received updated clock correlation parameters. New RDD parameters found in OPM-62s are stored in DMS for use during the following contact. In order to calculate RDD clock errors, the process obtains input, prior to the contact, in the way of Flight Dynamics Facility (FDF) Range data and stored RDD parameters from the Data Management Subsystem (DMS) . The Clock Correlation Analysis process generates products in the way of reports, Event Messages, and Command Requests. To generate Reports, previous Clock Correlation calculations are accessed through DMS. Reports and Event Messages containing information about the Spacecraft Clock Error are sent to DMS. Nominally, Command Requests will only need to be sent to the FOS User Interface (FUI) about once every seven days. When it is necessary to adjust the clock, a command table load will be generated by the Command Management Subsystem (CMS) and forwarded to FUI with the command request.

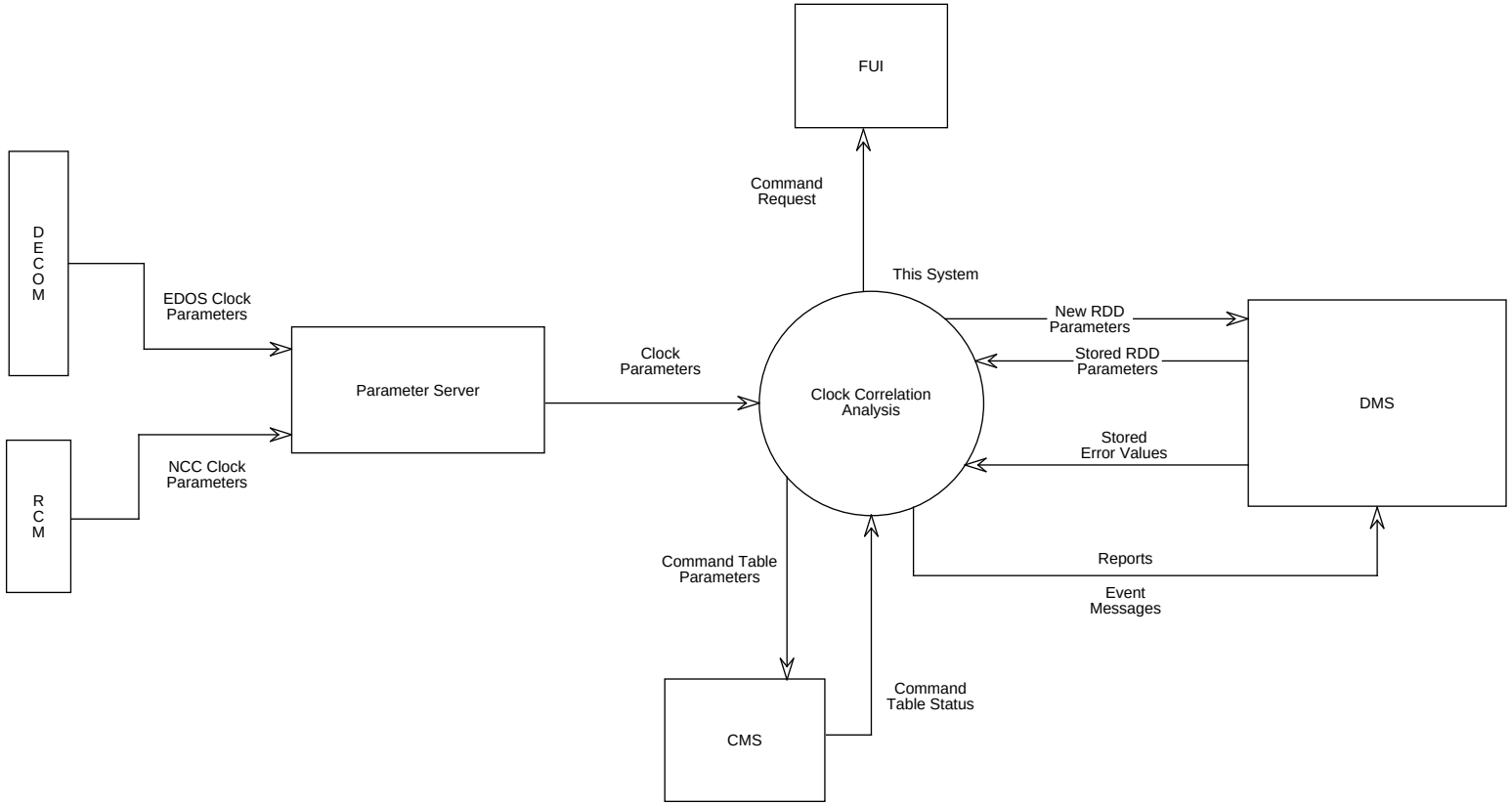

Figure 3.4-2. Clock Correlation Analysis Context Diagram

3.4.2 Clock Correlation Analysis Interfaces

***Table* 3.4.2. Clock Correlation Analysis Interfaces**

Interface Service	Interface Class	Interface Class Description	Service Provider	Service User	Frequency
Provide Process Parameters	ParameterMonitor	Class by which process receives parameters from DECOM and RCM	ANA/FUI	ANA	As often as parameter values are updated
Provide FDF Range Data	FoDsFile	Class through which data is obtained	DMS	ANA	Immediately prior to each contact
Provide Previous Clock Errors	FoDsFile	Class through which data is obtained	DMS	ANA	Immediately after each contact
Store Clock Reports	FoDsFile	Class through which data is stored	DMS	ANA	Immediately after each contact
Store RDD Parameters	FoDsFile	Class through which data is stored	DMS	ANA	Immediately after each contact
Build Clock Table Load	FmMsGenTable	Proxy class which generates a Command Table Load from Report contents	CMS	ANA	Whenever a command request is built
Provide Table Load generation Status	FmMsGenTable		CMS	ANA	Whenever a table load is built
Build Command Request	FuCrCmdRequestInfo	Contains clock adjustment commands	FUI	ANA	Whenever the clock needs adjustment
Generate Event Messages	FdEvEventLogger		DMS	ANA	Whenever a clock error is calculated

3.4.3 Clock Correlation Analysis Object Model

-Refer to Figure 3.4-3 Clock Correlation Analysis Object Model

When FaCcClockController is initiated, it initiates both FaCcRddError and FaCcUsccsError. When FaCcRddError is initiated, it accesses FDF range data from FoDsFile. Then, FaCcClockController accesses stored RDD parameters from FoDsFile, after which, calling the CalcVariables operation which propagates down to FaCcDelays, prompting it to update its process variables from those accessed parameters. Once a contact has begun, the ParameterMonitor

notifies FaCcClockController when it has received updated clock correlation parameters. There will be three distinct categories of parameters: those coming from a telemetry package, those coming from an OPM-62, and those coming from an OPM-66. FaCcClockController retrieves those updated parameters from the ParameterMonitor, and then either calls the CalcVariables operation which propagates down to its aggregate parts, or in the case of OPM-62 parameters coming after a contact, stores those parameters in FoDsFile. These aggregate parts, FaCcTlmEdu, FaCcDelays, and FaCcPnEpochPairs receive the propagated CalcVariables operation, prompting them to derive process variables from the updated parameter values, and store those variables in their respective containers. When this propagated operation is completed, FaCcClockController will either notify FaCcClockError that the process has received updated variable values (in the case where OPM-66 parameters were the ones that arrived), or it will become idle, awaiting another set of parameters from ParameterMonitor (in the case where telemetry parameters were the ones that arrived). FaCcClockController will notify FaCcClockError to calculate a clock error. For RDD calculation, this notification occurs once per minute, while for USCCS calculations, this notification occurs upon whenever an OPM-66 arrives. Once FaCcClockError has been notified, it then accesses updated process variables from the respective Containers of FaCcTlmEdu, FaCcDelays, and FaCcPnEpochPairs in order to begin its calculations. When FaCcClockError needs Spacecraft range estimates, it will request those values from FaCcRangeEst. In the case of a USCCS calculation, this gives a PN Epoch based range estimate via the Container of PN Epoch times, while in the case of an RDD calculation, FaCcFdfRange simply returns the previously retrieved FDF predicted range data. Both FaCcRddError and FaCcUsccsError will generate reports (sent to FoDsFile after the contact) and Event Messages (sent to FdEvEventLogger whenever an error is calculated) describing the Clock Correlation Analysis process results. FaCcClockError will generate and send to FuCrCmdRequestInfo a Command Request to adjust the Spacecraft Clock as often as the process deems necessary. To do this, FaCcClockError first sends FmMsGenTable table load parameters needed to build a command table load associated with the command request.

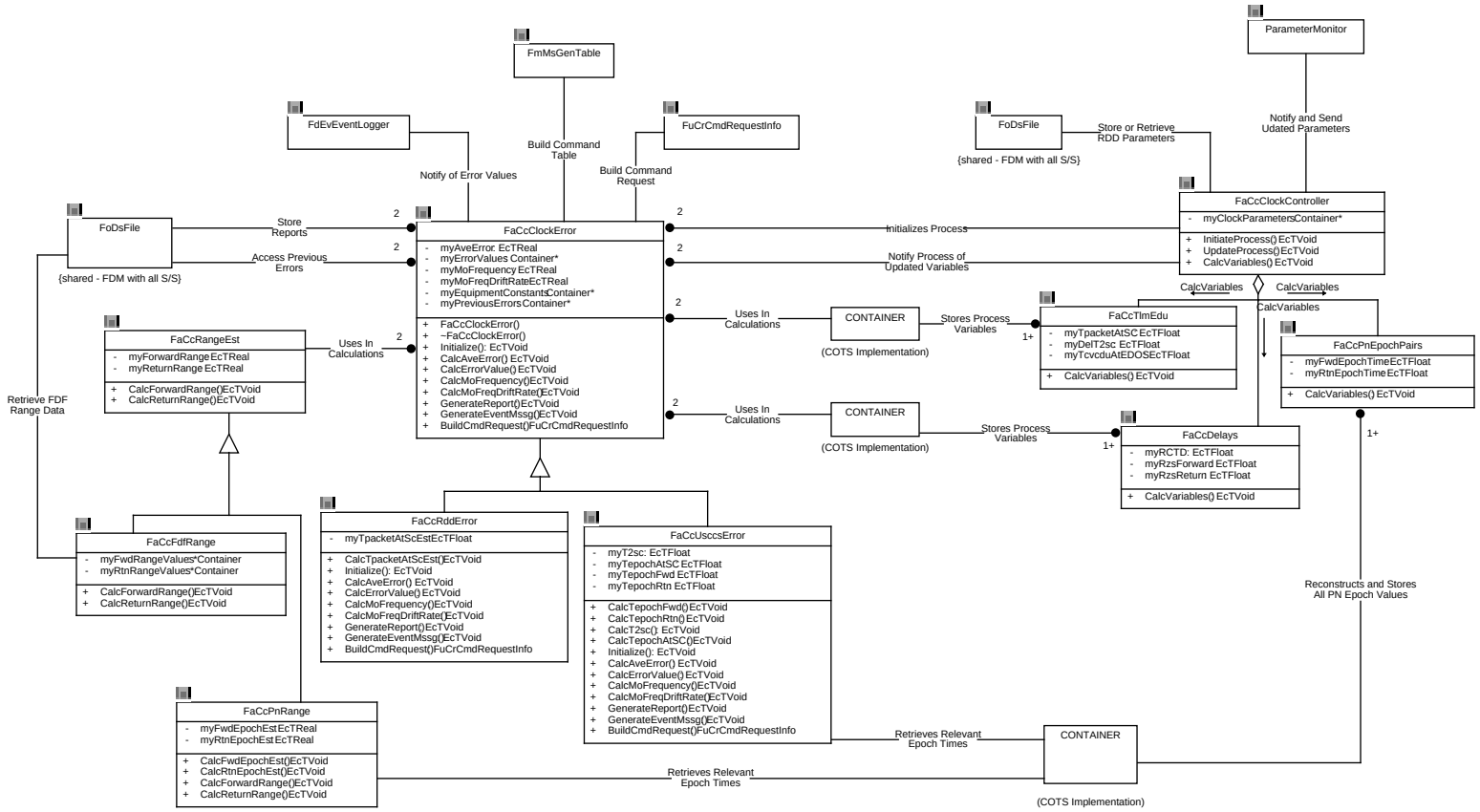

Figure 3.4-3. Clock Correlation Analysis Object Model

3.4.4 Clock Correlation Analysis Dynamic Model

3.4.4.1 USCCS Clock Correlation Processing Scenario Abstract

This section details the USCCS Clock Correlation process

3.4.4.1.1 USCCS Clock Correlation Processing Summary Information

Interfaces:

ParameterMonitor

FdEvEventLogger

FoDsFile

FmMsGenTable

FuCrCmdRequestInfo

Stimulus:

Arrival of OPM-66 parameters to ParameterMonitor

Desired Response:

Calculate Error

Generate Event Message

Generate & Store Report (after termination of contact)

Generate Table Load & Command Request (when necessary)

Pre-Conditions:

Telemetry Parameters Received From ParameterMonitor

Previous Errors Locatable

Post-Conditions:

Process Terminated

3.4.4.1.2 USCCS Clock Correlation Scenario Description

-Refer to Figure 3.4-4 USCCS Clock Correlation Processing Event Trace

The USCCS Clock Correlation Analysis process begins after a TDRSS Tracking Service when the ParameterMonitor receives OPM-66 parameters sent by the NCC. ParameterMonitor notifies FaCcClockController when it receives new USCCS parameters. This triggers FaCcClockController to call the operation CalcVariables which propagates down to its aggregate parts, FaCcDelays, FaCcTlmEdu, and FaCcPnEpoch. Once FaCcClockController has completed its propagated operation, it notifies FaCcUsccsError that the process variables have been updated. FaCcUsccsError then initiates the USCCS algorithm, accessing the updated variables from the respective Containers of FaCcDelays, FaCcTlmEdu, and FaCcPnEpochPairs. To calculate a Spacecraft Clock Error, FaCcUsccsError needs to access a PN Epoch based range estimate from FaCcPnRange, which in turn retrieves necessary information from the Container of PN Epoch times. Once the calculations have been made, an event message is sent to FdEvEventLogger, providing notification of the latest USCCS results. After a contact, previously calculated errors are access through FoDsFile, and are use to generate a report which is sent back to FoDsFile. To adjust

the Spacecraft Clock based upon its calculations, FaCcUsccsError must first generate a Table Load via FmMsGenTable. It will then generate a command request sent to FuCrCmdRequestInfo.

3.4.4.2 RDD Clock Correlation Processing Scenario Abstract

This section details the RDD Clock Correlation process.

3.4.4.2.1 RDD Clock Correlation Processing Summary Information

Interfaces:

ParameterMonitor

FdEvEventLogger

FoDsFile

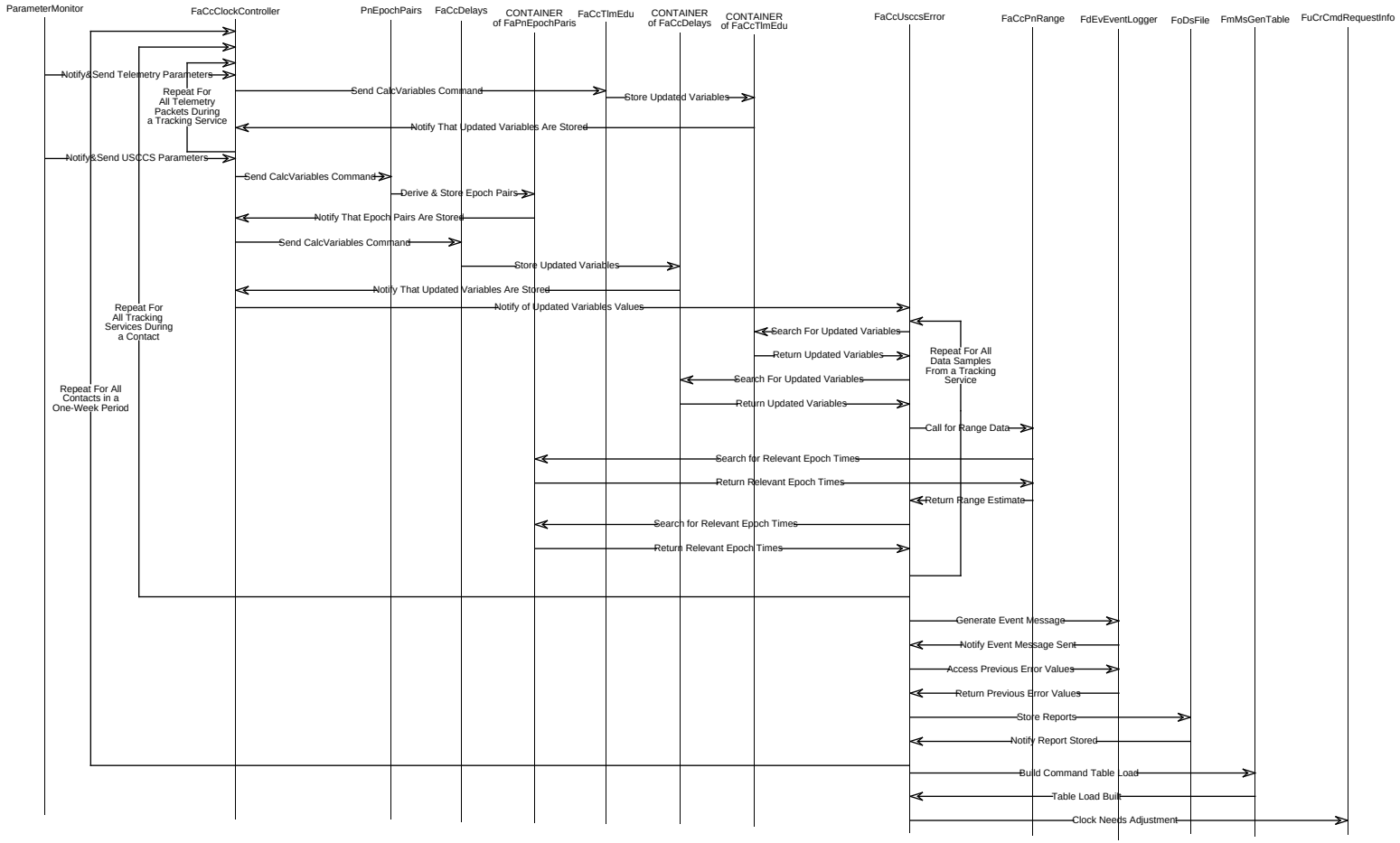

Figure 3.4-4. USCCS Clock Correlation Processing Event Trace

FmMsGenTable

FuCrCmdRequestInfo

Stimulus:

One Minute Elapsed During Real-Time Contact

Desired Response:

Calculate Error

Generate Event Message

Generate & Store Report (after termination of contact)

Generate Table Load & Command Request (when necessary)

Pre-Conditions:

Telemetry Parameters Received From ParameterMonitor

Previous Errors Locatable

Post-Conditions:

Process Terminated

3.4.4.2.2 RDD Clock Correlation Processing Scenario Description

-Refer to Figure 3.4-5 RDD Clock Correlation Processing Event Trace

The RDD Clock Correlation Analysis process begins at the start of a Real-Time contact. When FaCcClockController is initialized, it initializes FaCcRddError which prompts FaCcRddError to retrieve FDF range data from FoDsFile. Then, FaCcClockController retrieves stored RDD parameters. Upon retrieving those parameters, FaCcClockController call the CalcVariables operation, which is propagated down to its aggregate part FaCcDelays. Once FaCcDelays has completed its CalcVariables operation, FaCcClockController waits for Telemetry parameters. ParameterMonitor notifies FaCcClockController it begins receiving Telemetry parameters. This triggers FaCcClockController to call the operation CalcVariables which propagates down to its aggregate part FaCcTlmEdu. Once FaCcClockController has completed its propagated operation, it waits for the next arrival of updated Telemetry parameters. After Telemetry parameters have started arriving, every time a complete minute has elapsed it notifies FaCcRddError to begin its Clock Error calculations. FaCcRddError then initiates the RDD algorithm, accessing the updated variables from the respective Containers of FaCcDelays and FaCcTlmEdu. To calculate a Spacecraft Clock Error, FaCcRddError needs to access previously retrieved FDF Range data from FaCcFdfRange. Once the calculations have been made, an event message is sent to FdEvEventLogger, providing notification of the latest RDD results. After termination of a Real-Time contact, FaCcRddError first accesses previous Clock Error calculations via FoDsFile, in

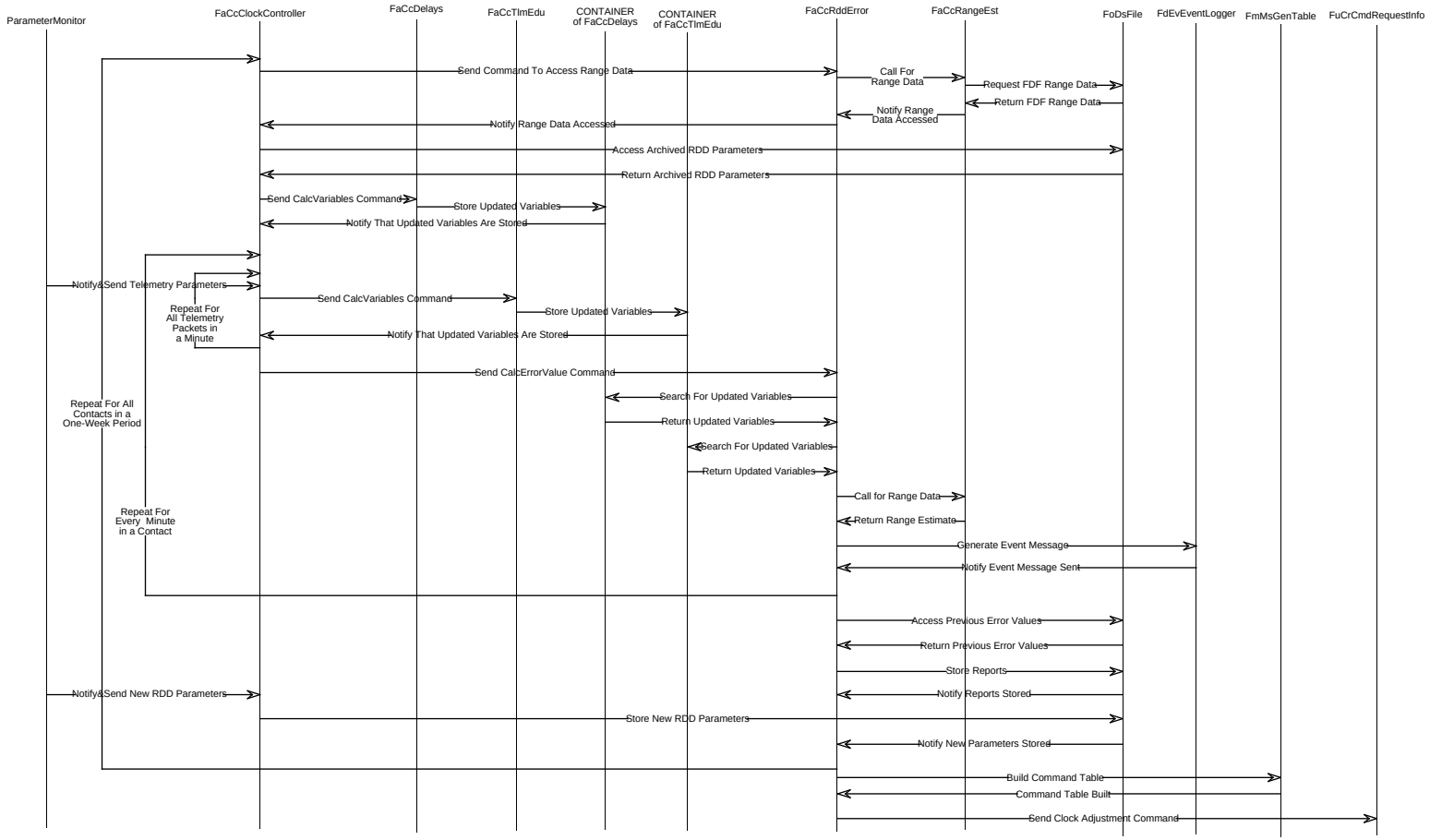

Figure 3.4-5. RDD Clock Correlation Processing Event Trace

order to generate a report. Once the report has been constructed, it is stored in DMS via FoDsFile. Based upon its calculations, FaCcRddError will generate a command table load and command request when necessary; however, nominally, the FaCcUsccsError generated command table load and command request will be sent to FuCrCmdRequestInfo and FmMsGenTable respectively, whereas the FaCcRddError generated command table load and command request will only be processed when a FaCcUsccsError command request is not available.

3.4.5 Clock Correlation Data Dictionary

FaCcClockController

class **FaCcClockController**

This class initiates both FaCcRddError and FaCcUsccsError. After it has done so, FaCcClockController accesses stored RDD parameters from FoDsFile, and calls the CalcVariables operation to update the process variables associated with those parameters. Then, it remains idle until it is notified by the ParameterMonitor when ParameterMonitor has received updated clock parameters. Once notified, it calls the CalcVariables member function which updates process variables associated with those parameters. Once those variables are updated, it notifies FaCcClockError that process variables have been updated, prompting the clock correlation calculations to begin. After a contact has terminated, FaCcClockController will wait for updated RDD parameters to arrive at the ParameterMonitor; when they have arrived, it will send those parameters to FoDsFile to be stored for use during the subsequent contact.

Public Functions

`EcTVoid` **`CalcVariables`**`(void)`

Updates the process variables related to the updated parameters

`EcTVoid` **`InitiateProcess`**`(void)`

Initiates FaCcClockError, accesses RDD parameters, calls UpdateProcess

`EcTVoid` **`UpdateProcess`**`(void)`

UpdateProcess

Receives notification and location of updated parameters, stores RDD parameters, calls CalcVariables, and when necessary, begins a clock error calculation by notifying FaCcClockError of updated variables

Private Data

`Container*` **`myClockParameters`**

A list of all clock process parameters from which process variables are determined. Its location is passed as an argument when ParameterMonitor calls UpdateProcess.

FaCcClockError

class **FaCcClockError**

This is the abstract superclass whose children classes calculate their respective clock errors, and generate their respective products

Public Construction

`FaCcClockError``(void)`

Constructor

This is the default constructor for FaCcClockError

`~FaCcClockError``(void)`

Destructor

This is the default destructor for FaCcClockError

Public Functions

`FuCrCmdRequestInfo` **`BuildCmdRequest`**`(void)`

Generates an instantiation of FuCrCmdRequestInfo at appropriate times based on the contents of the generated clock error report

`EcTVoid` **`CalcAveError`**`(void)`

Calculates the mean value of the contents of myErrorValues

`EcTVoid` **`CalcErrorValue`**`(void)`

Calculates the algorithm based clock error and stores it in myErrorValues

`EcTVoid` **`CalcMoFreqDriftRate`**`(void)`

Calculates the Master Oscillator's frequency drift rate, based on the previous and current value of myMoFrequency

`EcTVoid` **`CalcMoFrequency`**`(void)`

Calculates the Master Oscillator's frequency, based on the clock error

`EcTVoid` **`GenerateEventMssg`**`(void)`

GenerateEventMssg

Generates an Event Message notification of the current clock error

`EcTVoid` **`GenerateReport`**`(void)`

Generates a report containing myAveError, myMoFrequency, myMoFreqDriftRate and sends that report to FoDsFile

`EcTVoid` **`Initialize`**`(void)`

Performs all initialization required for a process to run

Private Data

`EcTReal` **`myAveError`**

The mean value of the contents of myErrorValues

`Container*` **`myEquipmentConstants`**

A list of the constant equipment delays specific to an algorithm

`Container*` **`myErrorValues`**

A list of calculated error values per Operational Message

`EcTReal` **`myMoFreqDriftRate`**

The rate of change of myMoFrequency based on previous and current values of myMoFrequency

`EcTReal` **`myMoFrequency`**

The calculated Master Oscillator frequency, based on the clock error

FaCcDelays

`class` **`FaCcDelays`**

This class accesses updated clock parameters from FaCcClockController and uses them to derive and store the measured delays needed to derive a clock error

Public Functions

`EcTVoid` **`CalcVariables`**`(void)`

Calculates and Stores the updated values of myRCTD, myRzsForward, and myRzsReturn

Private Data

`EcTFloat` **`myRCTD`**

Return Channel Time Delay; the measured time delay experienced by a telemetry signal as it passes through WSGTU/

STGT

`EcTFloat` **`myRzsForward`**

Forward Range Zero Set value; the measured time delay experienced by a forward PN Epoch as it passes through TDRSS and WSGTU/STGT

`EcTFloat` **`myRzsReturn`**

Return Range Zero Set value; the measured time delay experienced by a return PN Epoch as it passes through TDRSS and WSGTU/STGT

FaCcFdfRange

`class` **`FaCcFdfRange`**

This is the child class of FaCcRangeEst which accesses FDF range data from FoDsFile specifically for the RDD method

Base Classes

`public` **`FaCcRangeEst`**

Public Functions

`EcTVoid` **`CalcForwardRange`**`(void)`

Calculate the value of myForwardRange

`EcTVoid` **`CalcReturnRange`**`(void)`

Calculate the value of myReturnRange

FaCcPnEpochPairs

`class` **`FaCcPnEpochPairs`**

This class accesses updated clock parameters from FaCcClockController and uses them to derive the recorded Forward and Return PN Epoch times. Once derived, these time values are used to reconstruct all the unrecorded PN Epoch times. Finally, all the new PN Epoch times are stored in a Container.

Public Functions

`EcTVoid` **`CalcVariables`**`(void)`

Calculates the updated values of myFwdEpochTime and myRtnEpochTime, then reconstructs all unrecorded PN Epoch times, finally storing all time values in a Container

Private Data

`EcTFloat` **`myFwdEpochTime`**

The recorded Forward Delta Time in an OPM-66; the recorded departure time of a Forward PN Epoch from WSGTU/STGT

`EcTFloat` **`myRtnEpochTime`**

The recorded Return Delta Time in an OPM-66; the recorded arrival time of a return PN Epoch at WSGTU/STGT

FaCcPnRange

`class` **`FaCcPnRange`**

This is the child class of FaCcRangeEst which accesses PN Epoch times from their Container, and uses those times to derive a range estimate

Base Classes

`public` **`FaCcRangeEst`**

Public Functions

`EcTVoid` **`CalcForwardRange`**`(void)`

Use the values of myFwdEpochEst and myRtnEpochEst to derive a forward range estimate

`EcTVoid` **`CalcFwdEpochEst`**`(void)`

Calculate the updated value of myFwdEpochEst

`EcTVoid` **`CalcReturnRange`**`(void)`

Use the values of myRtnEpochEst and myFwdEpochEst to derive a return range estimate

`EcTVoid` **`CalcRtnEpochEst`**`(void)`

Calculate the updated value of myRtnEpochEst

Private Data

`EcTReal` **`myFwdEpochEst`**

A Forward Epoch time associated with a telemetry time stamp

`EcTReal` **`myRtnEpochEst`**

A Return Epoch time associated with a telemetry time stamp

FaCcRangeEst

`class` **`FaCcRangeEst`**

This is the abstract parent class whose child classes derive Spacecraft Ranges with their respective methods for their respective related clock error algorithms

Public Functions

`EcTVoid` **`CalcForwardRange`**`(void)`

Calculate the value of myForwardRange

`EcTVoid` **`CalcReturnRange`**`(void)`

Calculate the value of myReturnRange

Private Data

`EcTReal` **`myForwardRange`**

The distance traveled by a signal from the ground to the Spacecraft

`EcTReal` **`myReturnRange`**

The distance traveled by a signal from the Spacecraft to the Ground

FaCcRddError

`class` **`FaCcRddError`**

This is the child class of FaCcClockError that uses the RDD algorithm to derive a clock error

Base Classes

`public` **`FaCcClockError`**

Public Functions

`FuCrCmdRequestInfo` **`BuildCmdRequest`**`(void)`

Generates an instantiation of FuCrCmdRequest at appropriate times based on the contents of the generated clock error

report

`EcTVoid` **`CalcAveError`**`(void)`

Calculates the mean value of the contents of myErrorValues

`EcTVoid` **`CalcErrorValue`**`(void)`

Calculates the algorithm based clock error and stores it in myErrorValues

`EcTVoid` **`CalcMoFreqDriftRate`**`(void)`

Calculates the Master Oscillator's frequency drift rate, based on the previous and current value of myMoFrequency

`EcTVoid` **`CalcMoFrequency`**`(void)`

Calculates the Master Oscillator's frequency, based on the clock error

`EcTVoid` **`CalcTpacketAtScEst`**`(void)`

Calculate the value of myTpacketAtScEst

`EcTVoid` **`GenerateEventMssg`**`(void)`

GenerateEventMsg

Generates an Event Message notification of the current clock error

`EcTVoid` **`GenerateReport`**`(void)`

Generates a report containing myAveError, myMoFrequency, myMoFreqDriftRate and sends that report to FoDsFile

`EcTVoid` **`Initialize`**`(void)`

Access FDF Range Data from FoDsFile by initiating FaCcFdfRange

Private Data

`EcTFloat` **`myTpacketAtScEst`**

A calculated UTC estimate of the time of a telemetry packet's construction

FaCcTlmEdu

`class` **`FaCcTlmEdu`**

This class accesses updated clock parameters from FaCcClockController and uses them to derive and store telemetry specific clock variables

Public Functions

`EcTVoid` **`CalcVariables`**`(void)`

Calculates and stores the updated values of myTpacketAtSC, myDelT2sc, and myTcvcduAtEDOS

Private Data

`EcTFloat` **`myDelT2sc`**

The spacecraft's reading of when a Forward PN Epoch arrived at the spacecraft

`EcTFloat` **`myTcvcduAtEDOS`**

The time when a CVCDU coded telemetry stream arrives at EDOS to be decoded

`EcTFloat` **`myTpacketAtSC`**

The time when a decoded telemetry packet has been reconstructed at EDOS

FaCcUsccsError

`class` **`FaCcUsccsError`**

This is the child class of FaCcClockError that uses the USCCS algorithm to derive a clock error

Base Classes

`public` **`FaCcClockError`**

Public Functions

`FuCrCmdRequestInfo` **`BuildCmdRequest`**`(void)`

Generates an instantiation of FuCrCmdRequest at appropriate times based on the contents of the generated clock error report

`EcTVoid` **`CalcAveError`**`(void)`

Calculates the mean value of the contents of myErrorValues

`EcTVoid` **`CalcErrorValue`**`(void)`

Calculates the algorithm based clock error and stores it in myErrorValues

`EcTVoid` **`CalcMoFreqDriftRate`**`(void)`

Calculates the Master Oscillator's frequency drift rate, based on the previous and current value of myMoFrequency

`EcTVoid` **`CalcMoFrequency`**`(void)`

Calculates the Master Oscillator's frequency, based on the clock error

`EcTVoid` **`CalcT2sc`**`(void)`

Calculates the value of T2sc

`EcTVoid` **`CalcTepochAtSC`**`(void)`

Calculates the value of TepochAtSC

`EcTVoid` **`CalcTepochFwd`**`(void)`

Calculate the value of myTepochFwd

`EcTVoid` **`CalcTepochRtn`**`(void)`

Calculates the value of myTepochRtn

`EcTVoid` **`GenerateEventMssg`**`(void)`

GenerateEventMsg

Generates an Event Message notification of the current clock error

`EcTVoid` **`GenerateReport`**`(void)`

Generates a report containing myAveError, myMoFrequency, myMoFreqDriftRate and sends that report to FoDsFile

Private Data

`EcTFloat` **`myT2sc`**

The Spacecraft time when a PN Epoch arrived at the Spacecraft

`EcTFloat` **`myTepochAtSC`**

The UTC time when a PN Epoch arrived at the Spacecraft

`EcTFloat` **`myTepochFwd`**

The recorded departure time of a PN Epoch associated with a particular telemetry parameter

`EcTFloat` **`myTepochRtn`**

The recorded departure time of a PN Epoch associated with a particular telemetry parameter

3.5 Solid State Recorder Analysis Processing

The SSR analysis process is part of the real time string which monitors the SSR and detects faults that affect a playback. The process evaluates SSR housekeeping telemetry, EDOS SSR CODA data and NCC Link status data to ensure the receipt of SSR data. It identifies and reports problems such as loss of data identified in EDOS CODA data and loss of signal identified in NCC ODM data and provides recommended solutions to the problems to be forwarded to the user. The SSR Analysis Processing uses a suite of RTworks COT tools to receive data parameters, analyze and forward recommendations to identified problems. The RTworks Tools also produces a SSR report identifying the state of the SSR at the end of a contact and forwards the report to PAS and FUI.

3.5.1 Solid State Recorder Analysis Processing Context

-Refer to Figure 3.5-1 Solid State Record Analysis Processing Context Diagram

The Analysis Subsystem SSR management process has an interface with PAS in order to receive predicted SSR buffer information and a schedule of spacecraft contacts. The SSR process receives real time spacecraft and ground telemetry from the parameter server within the TLM subsystem. This telemetry is used along with the PAS information to monitor SSR replays and detect errors. For error reporting and resolution, there is an interface with FUI, through which anomaly reports and recommended procedures are distributed to the user.

The SSR subsystem consists of two parts: The RT-Works COTS product, which contains the algorithms for error detection and resolution, and the FaRtRTWorksDataServer, which provides and interface between telemetry and RT-Works.

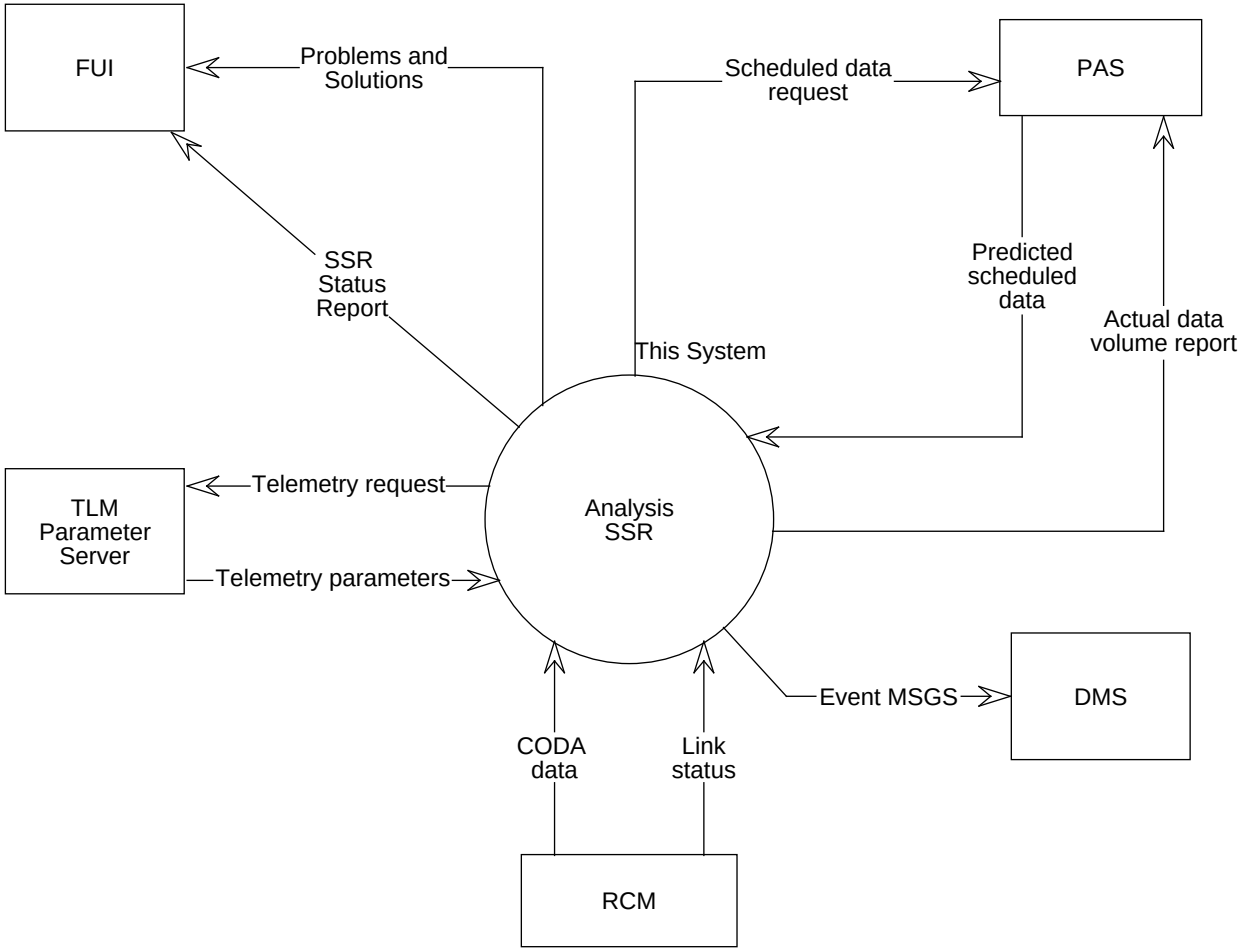

Figure 3.5-1. Solid State Recorder Analysis Processing Context

3.5.2 Solid State Recorder Analysis Processing Interfaces

***Table* 3.5.2. Solid State Recorder Analysis Processing Interfaces**

Interface Service	Interface Class	Interface Class Description	Service Provider	Service User	Frequency
Sends User Parameters	Parameter Server	Provide Parameter Monitor w/User Parameters	Decom/ TLM	ANA	SSR parameters every second during playback
Sends Request for Schedule Information Receives Schedule Information	FaSrContact	Request/Receive Schedule info Contacts/PB Data volume	Planning and Scheduling	ANA	Scheduling info Start/End of each contact
Sends SSR Parameters	FaRtRTWorksDataServer	Send TLM, ODM and CODA parameters to RTworks	ANA	ANA	Every contact TLM every second, CODA /ODM every five seconds
Sends Problem Descriptions and Recommendations	FoSrData	Send problem description, recommendation	ANA	FUI	Every contact when problems are detected and at end of contact
Sends Command Requests	FuCrCmdRequestInfo	Command re-quest for problem solution	FUI	ANA	Whenever recommend-ed procedures involve a com-mand
Sends SSR Status Reports	FaSrPbackReport	SSR status report	ANA	FUI, PAS	After every SSR playback

3.5.3 SSR Analysis Processing Object Model

-Refer to Figure 3.5-2 Solid State Record Analysis Processing Object Model

The SSR Analysis object diagram shows the relationship between the RT-Works COTS product and the classes which assist in the SSR analysis processing. The FaSrRTworksDataServer will gather all required ground and spacecraft telemetry and send to the RT-Works COTS product so that it may be used to perform SSR Analysis. The RT-Works COTS product interfaces with planning an scheduling via the FaSrPbackReport for sending playback results, and the FaSrContact for sending predicted data ANA.

Finally, problems and solutions are sent to FUI via the FoSrData, and command requests are submitted via the FuCrCmdRequestInfo class.

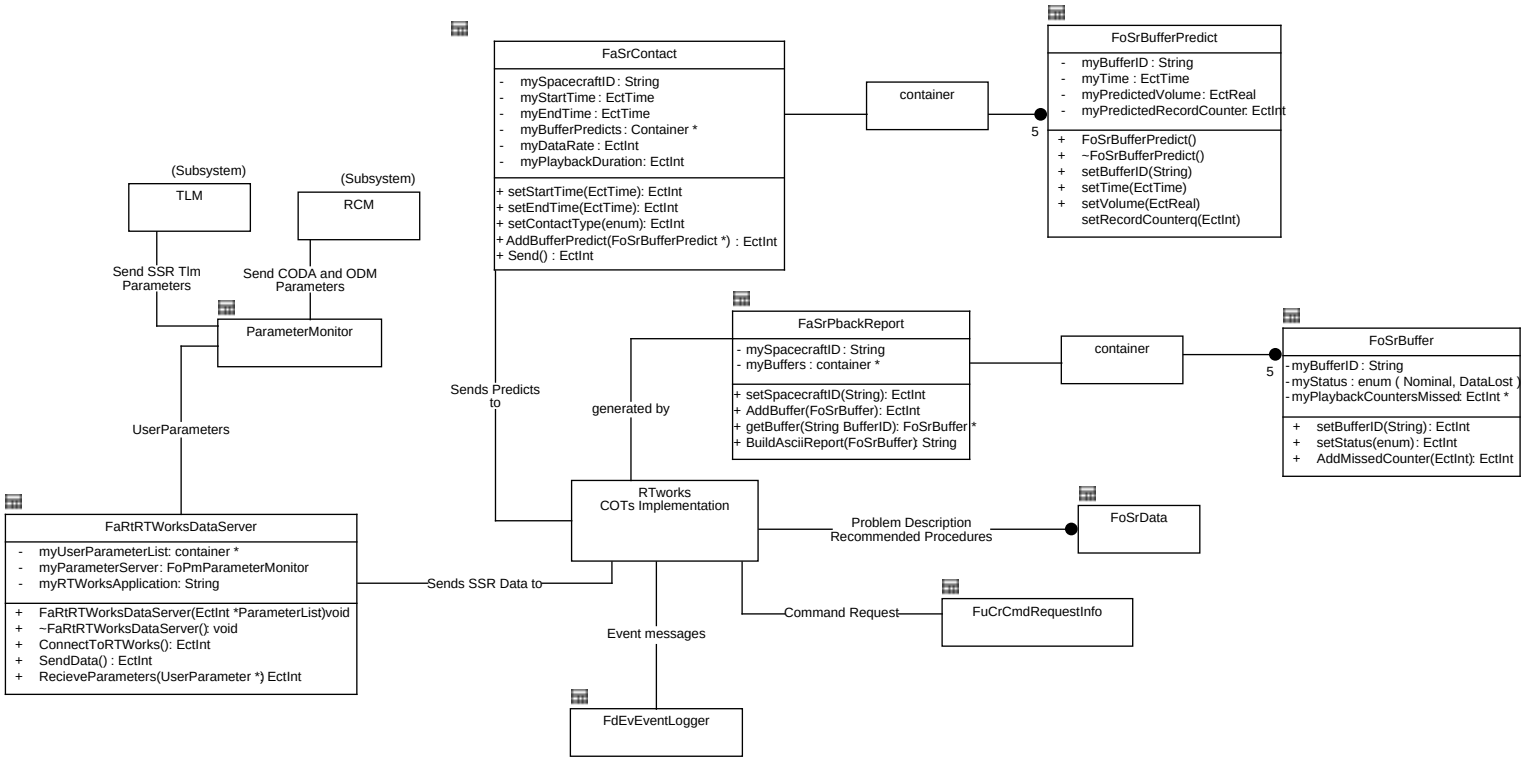

Figure 3.5-2. Solid State Recorder Analysis Processing Object Model

3.5.4 Solid State Recorder Analysis Processing Dynamic Model

The FaRtRTWorksDataServer obtains real-time SSR telemetry data , EDOS CODA and NCC ODM data from the TLM and RCM subsystems via the Parameter Monitor. The RTworks RTie Tool receives predicted schedule information from PAS . The FaRtRTWorksDataServer sends real-time telemetry, NCC and EDOS data to RTworks for analysis. The RTworks RTie Inference Engine Tool analyzes the SSR related data for anomalies and checks the SSR buffer counters. If there are any anomalies, the RTworks Rtie Tool sends a problem description and a recommendation to FUI and sends an event message to DMS. The RTie Tool also sends SSR buffer status reports to the PAS Subsystem and to FUI subsystems. The RTie Tool is composed of an Object Database, IF-THEN-ELSE rules, and user-defined external functions.

3.5.4.1 SSR Playback Nominal Scenario

3.5.4.1.1 SSR Playback Nominal Case Scenario Abstract

The purpose of the SSR Nominal scenario is to describe the SSR analysis process when no problems are encountered during a playback.

-Refer to Figure 3.5-3 SSR Playback Nominal Event Trace

3.5.4.1.2 SSR Playback Nominal Case Summary Information

Interfaces:

User Interface

Telemetry

Resource Management

Planning and Scheduling

DMS

ANA

Stimulus:

Start of a contact . Request and Receive playback schedule data, TLM parameters, EDOS SSR CODA data and NCC ODM Link Status data and forward data to RTworks.

Desired Response:

None (RTworks rules did not detect any problems during SSR playback.)

Pre-Conditions:

The FaRtRTWorksDataServer and RTworks Tools are initialized. The FaRtRTWorksDataServer is a member of a real time string and requests; playback schedule data from PAS, SSR TLM parameters, EDOS SSR CODA parameters, and NCC link Status parameters from RCM via the Parameter Monitor.

Post-Conditions:

The RTie Tool sends PAS and FUI a SSR status report identifying the state of the SSR at the end of a playback.

3.5.4.1.3 SSR Playback Nominal Case Scenario Description

Before the start of a contact SSR requests and receives playback schedule data from PAS, and requests TLM parameters from the TLM subsystem. SSR also requests EDOS SSR CODA data and NCC ODM Link Status data from the RCM subsystem. During a playback SSR receives TLM parameters, EDOS SSR CODA data and NCC ODM Link Status data and forwards the data to RTworks. RTworks rules did not detect any problems during SSR playback. At the end of the contact RTWorks sends PAS and FUI a SSR status report identifying the state of the SSR at the end of a playback.

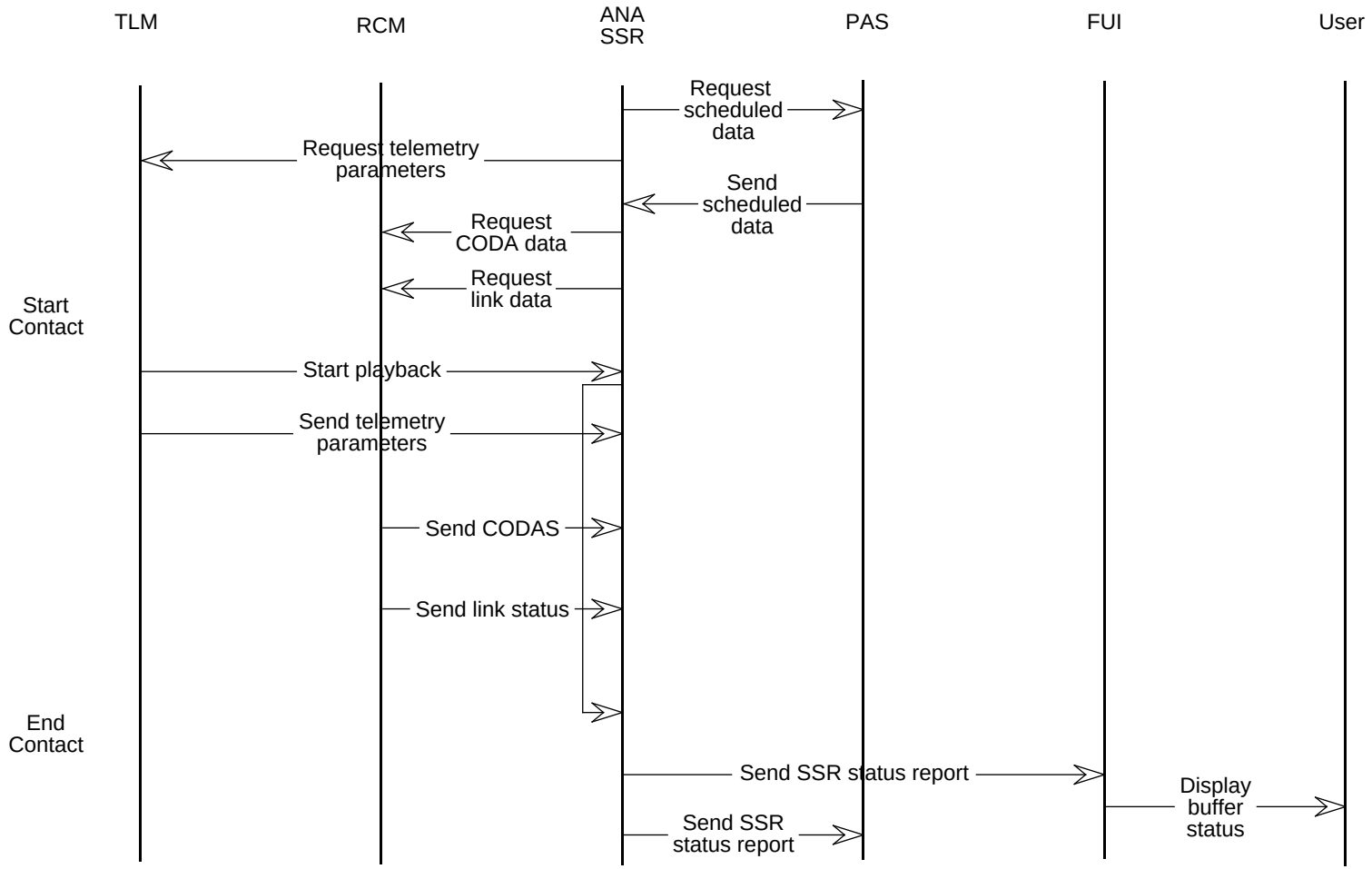

Figure 3.5-3. SSR Playback Nominal Event Trace

3.5.4.2 SSR Replay Loss of Data Replay NOT in Current Contact Scenario

3.5.4.2.1 SSR Replay Loss of Data - Replay NOT in Current Contact Scenario Abstract

- Refer to Figure 3.5-4 SSR Replay Loss of Data Replay NOT in Current Contact Event Trace

The purpose of the SSR Loss of Data scenario is to describe the SSR analysis process when data loss is encountered during a playback and an additional contact is required to prevent a buffer overflow.

3.5.4.2.2 SSR Playback Loss of Data - Replay NOT in Current Contact Summary Information

Interfaces:

User Interface

Telemetry

Resource Management

Planning and Scheduling

DMS

ANA

Stimulus:

Start of a contact . Request and receive playback schedule data, TLM parameters, **EDOS SSR CODA data indicating loss of data** and NCC ODM Link Status data. Forward the data to RTworks.

Desired Response:

The RTworks RTie Tool rules detect missing data from CODA data during the SSR playback and forwards a description of the problem to FUI and sends an event to DMS. After a playback the RTie Tool sends the SSR status report to FUI and Planning and Scheduling and request predicted schedule data from PAS. PAS 's response indicates that a buffer overflow will occur . RTie rules will detect the buffer overflow and forward a recommendation , indicating an additional contact is required to perform a playback to obtain the missing data. The recommendation is based on the data volume scheduled to be recorded and will indicate when the contact should be done to ensure there is no loss of data.

Pre-Conditions:

The FaRtRTWorksDataServer and RTworks Tools are initialized. The FaRtRTWorksDataServer is a member of a real time string and requests; playback schedule data from PAS, SSR TLM parameters, EDOS SSR CODA parameters, and NCC link Status parameters from RCM via the Parameter Monitor.

Post-Conditions:

The Rtie Tool sends FUI and PAS a SSR status report identifying the state of the SSR at the end of a playback. PAS send predicted schedule data which indicates buffer overflow condition. A recommendation to schedule another contact is sent to FUI.

3.5.4.2.3 SSR Playback Loss of Data - Replay NOT in Current Contact Scenario Description

Before the start of a contact SSR requests and receives playback schedule data from PAS, and requests TLM parameters from the TLM subsystem. SSR also requests EDOS SSR CODA data and NCC ODM Link Status data from the RCM subsystem. During a playback SSR receives TLM parameters, EDOS SSR CODA data and NCC ODM Link Status data and forwards the data to RTworks. During the SSR playback the RTworks rules detect missing data and keeps track of the missing data. At the end of the contact RTWorks sends PAS and FUI a SSR status report identifying the state of the SSR at the end of a playback. SSR request updated predicted schedule data from PAS. The predicted schedule data indicates that a buffer overflow condition exits. A recommendation to schedule another contact is sent to FUI.

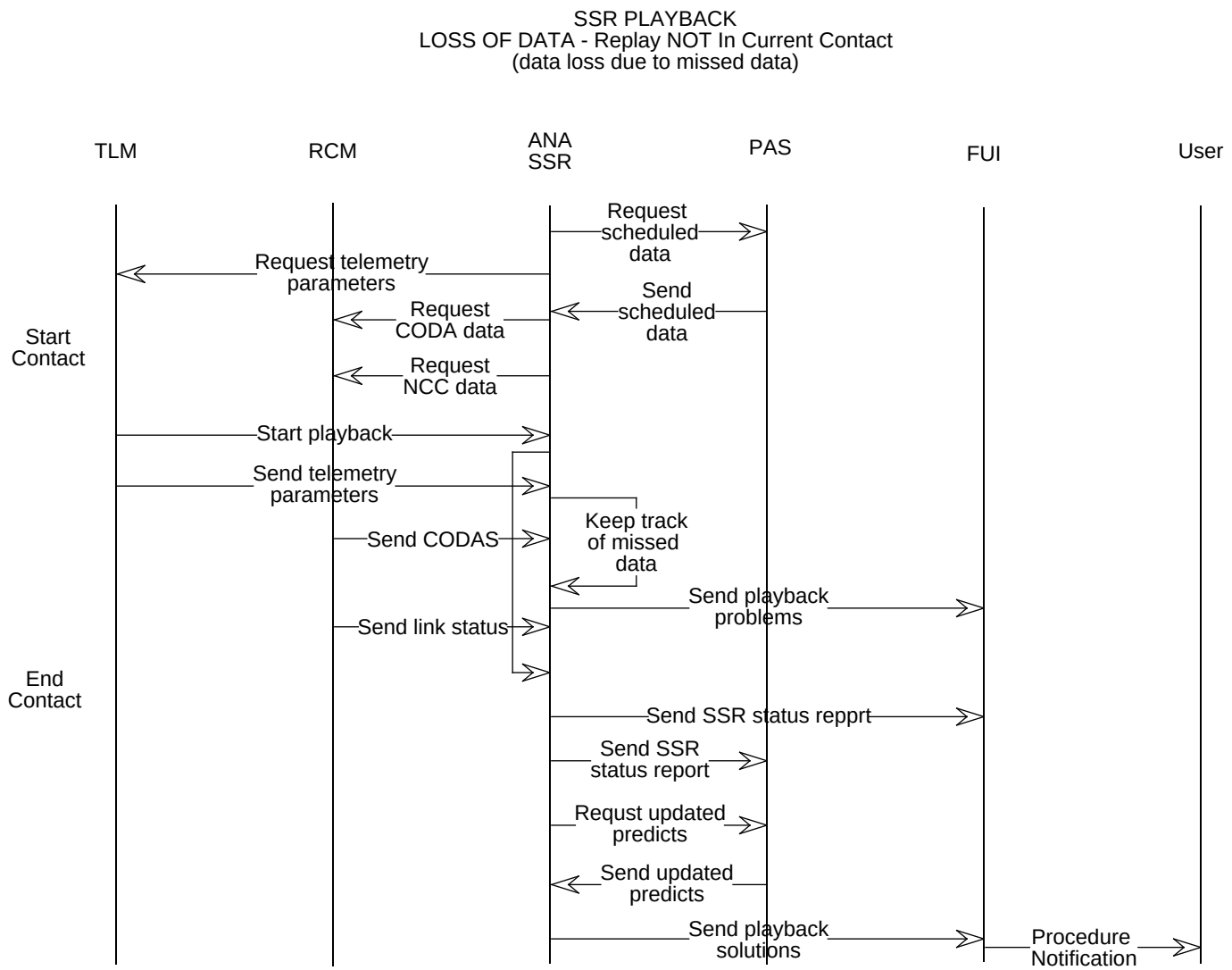

Figure 3.5-4. SSR Replay Loss of Data Replay NOT in Current Contact Event Trace

3.5.4.3 SSR Replay Loss of Data Replay During Current Contact Scenario

3.5.4.3.1 SSR Replay Loss of Data - Replay During Current Contact Scenario Abstract

- Refer to Figure 3.5-5 SSR Replay Loss of Data Replay During Current Contact Event Trace

The purpose of the SSR Loss of Signal scenario is to describe the SSR analysis process when a loss of signal is encountered during a playback and a replay is completed during the current contact.

3.5.4.3.2 SSR Playback Loss of Data - Replay During Current Contact Summary Information

Interfaces:

User Interface

Telemetry

Resource Management

Planning and Scheduling

DMS

ANA

Stimulus:

Start of a contact. Request and receive playback schedule data, TLM parameters, EDOS SSR CODA data and **NCC ODM Link Status indicates loss of signal** and forward data to RTworks.

Desired Response:

RTie rules detected a loss of signal from the ODM link status data during SSR playback . RTie forwards a description of the problem and the recommendation , in the form of a command procedure to re-acquire signal and playback missing data. The recommendation for the playback command is based on the data volume to be played back and time remaining in the scheduled contact.

Pre-Conditions:

The FaRtRTWorksDataServer and RTworks Tools are initialized. The FaRtRTWorksDataServer is a member of a real time string and requests; playback schedule data from PAS, SSR TLM parameters, EDOS SSR CODA parameters, and NCC link Status parameters from RCM via the Parameter Monitor.

Post-Conditions:

The Rtie Tool sends FUI and PAS a SSR status report identifying the state of the SSR at the end of a playback.

3.5.4.3.3 SSR Playback Loss of Data - Replay in Current Contact Scenario Description

Before the start of a contact SSR requests and receives playback schedule data from PAS, and requests TLM parameters from the TLM subsystem. SSR also requests EDOS SSR CODA data and NCC ODM Link Status data from the RCM subsystem. During a playback SSR receives TLM parameters, EDOS SSR CODA data and NCC ODM Link Status . RTworks rules detected a loss of signal from the ODM link status data during SSR playback . RTworks forwards a description of the problem and the recommendation , in the form of a command procedure to re-acquire signal and playback missing data. The recommendation for the playback command is based on the data volume to be played back and time remaining in the scheduled contact . At the end of the contact RTWorks sends PAS and FUI a SSR status report identifying the state of the SSR at the end of a playback.

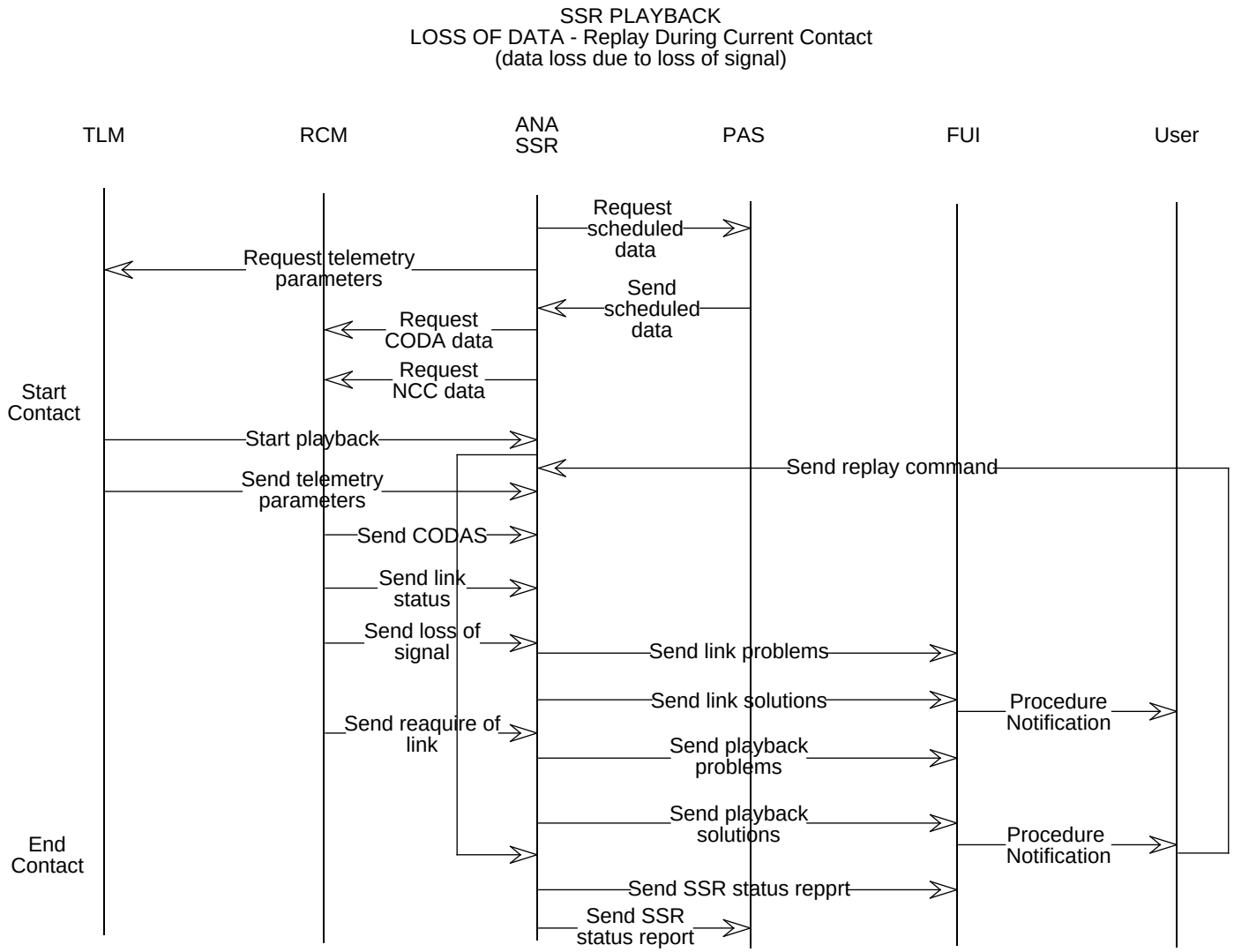

Figure 3.5-5. SSR Replay Loss of Data Replay During Current Contact Event Trace

3.5.4.4 Solid State Recorder Analysis Processing State Diagram

- Refer to Figure 3.5-6 Solid State Recorder Analysis Processing State Diagram

The COTS product shoden to implement SSR analysis is a rule based expert system. The state diagram does not represent a class, as it does with most state diagrams in an OMT design, but represents dynamic behavior of the rule based expert system. The state diagram covers cases from nominal to playback not initiated. This model will be the basis of rule creation and will be modified as development proceeds and more expert knowledge is gained.

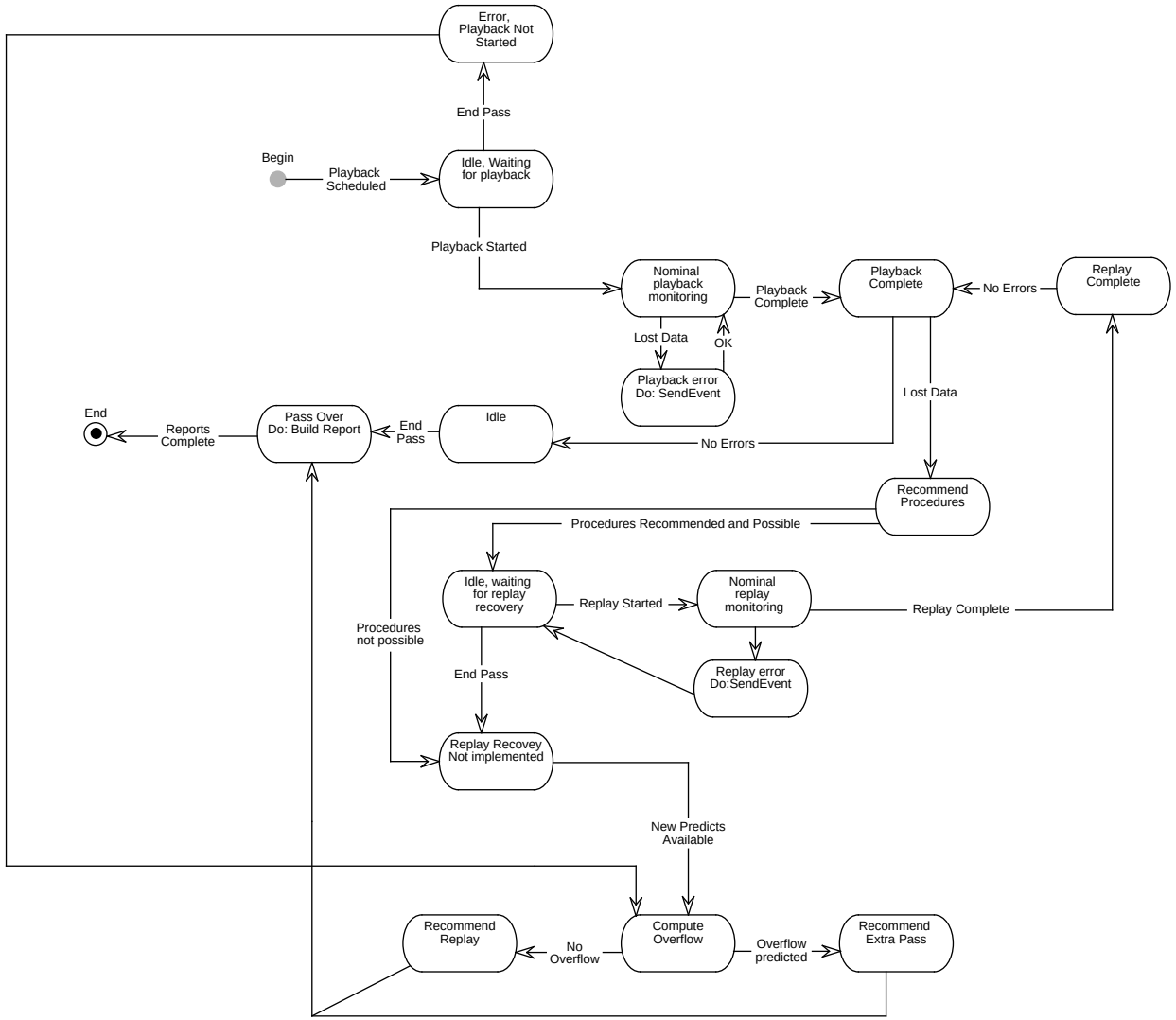

Figure 3.5-6. Solid State Recorder Analysis Processing State Diagram

3.5.5 Solid State Recorder Analysis Processing Data Dictionary

FaRtRTWorksDataServer

class **FaRtRTWorksDataServer**

Public Construction

```
void FaRtRTWorksDataServer(EctInt* ParameterList)
void ~FaRtRTWorksDataServer(void)
```

Public Functions

`EctInt` **`ConnectToRTWorks`**`(void)`

ConnectToRTWorks()

Function that registers with RTWorks to allow data passing between this application and RT-Works

`EctInt` **`RecieveParameters`**`(UserParameter*)`

RecieveParameters() Function that receives telemetry parameters from the Parameter monitor

`EctInt` **`SendData`**`(void)`

SendData()

Function that send data to RT-Works via RT-Works provided function call

Private Data

`FoPmParameterMonitor` **`myParameterMonitor`**

FOS class used to access telemetry parameters

`String` **`myRTWorksApplication`**

String needed by RTWorks function call to identify destination of data.

`container*` **`myUserParameterList`**

A list of user parameters that will be sent to RT-Works

FaSrContact

class **FaSrContact**

Private Functions

`EctInt` **`AddBufferPredict`**`(FoSrBufferPredict*)`

AddBufferPredict(FoSrBufferPredict *)

This operation adds a buffer predict to the list of buffer predicts

`EctInt` **`Send`**`(void)`

Send()

This function sends the predicted contact information to the analysis subsystem

```
EctInt setContactType(enum)
EctInt setEndTime(EctTime)
```

Private Data

`Container*` **`myBufferPredicts`**

A list of predicted volumes and record counters with an entry for each buffer

`EctInt` **`myDataRate`**

The rate of data being played back.

`EctTime` **myEndTime**

Time of the end of the contact

`EctInt` **myPlaybackDuration**

predicted playback duration in seconds

`String` **mySpacecraftID**

String identifier for spacecraft

`EctTime` **myStartTime**

Time of the start of the contact

FaSrPbackReport

class **FaSrPbackReport**

Public Functions

`EctInt` **AddBuffer**`(FoSrBuffer*)`
`FoSrBuffer*` **getBuffer**`(String BufferID)`

AddBuffer()

Adds a FoSrBuffer to the list of buffer status

`String` **BuildAsciiReport**`(FoSrBuffer*)`

BuildAsciiReport()

Builds an ASCII report from FoSrBuffers

`EctInt` **setSpacecraftID**`(String)`

Private Data

`container*` **myBuffers**

List containing status of each buffer

`String` **mySpacecraftID**

String identifier for spacecraft

FoSrBuffer

class **FoSrBuffer**

Public Functions

`EctInt` **AddMissedCounter**`(EctInt)`

AddMissedCounter(EctInt)

Adds a value, representing the Playback Counter value of missed data from the playback

`EctInt` **setBufferID**`(String)`
`EctInt` **setStatus**`(enum)`

Private Functions

`enum(Nominal, DataLost)`

indicates whether data was lost during the playback of this buffer

Private Data

`String` **myBufferID**

ID indicating which buffer this is. Option could be Aster, MISR, etc.., depending on mission and SC config

`EctInt* myPlaybackCountersMissed`

An array of integers showing which Playback Counters were missed during the playback. Each entry in the array represents one counter

FoSrBufferPredict

class **FoSrBufferPredict**

Public Construction

FoSrBufferPredict(void)
~FoSrBufferPredict(void)

Public Functions

void **setBufferID**(String)
void **setRecordCounter**(EctInt)
void **setTime**(EctTime)
void **setVolume**(EctReal)

Private Data

`String myBufferID`

String indicating which buffer this is. Examples would be ASTER, MISR, etc.. depending on the mission and the SSR configuration

`EctInt myPredictedRecordCounter`

predicted record counter

`EctReal myPredictedVolume`

predicted volume of the buffer

`EctTime myTime`

Time of the buffer predict

3.6 User Algorithms

The User Defined Algorithms is an important part of the analysis process. It allows the user to write complicated and time consuming functions and apply them to spacecraft telemetry in off-line analysis. As the spacecraft matures and changes over time, new problems will arise, many of which can be dealt with by creating new algorithms and adding them to existing analysis requests. This has the benefit of not requiring a new delivery of the EOC system software, and therefore has a turnaround time of several days, the exact duration depending on the method of acceptance by the user of the new algorithm.

3.6.1 User Algorithms Context

User Algorithms are generated by the user using a text/programming editor. They are written in the C or C++ language, and compiled into shareable objects. The algorithm is then registered with the system via the User Interface. This registration contains the names of the input and output variables within the User Algorithm, as well as the name and path of the shared object. The algorithm is requested within an analysis request, along with the parameter mnemonics for each input and output. When the request is processed, the algorithm us linked into the analysis process, and invoked using the criteria specified in the request.

- Refer to Figure 3.6-1 User Algorithms Context

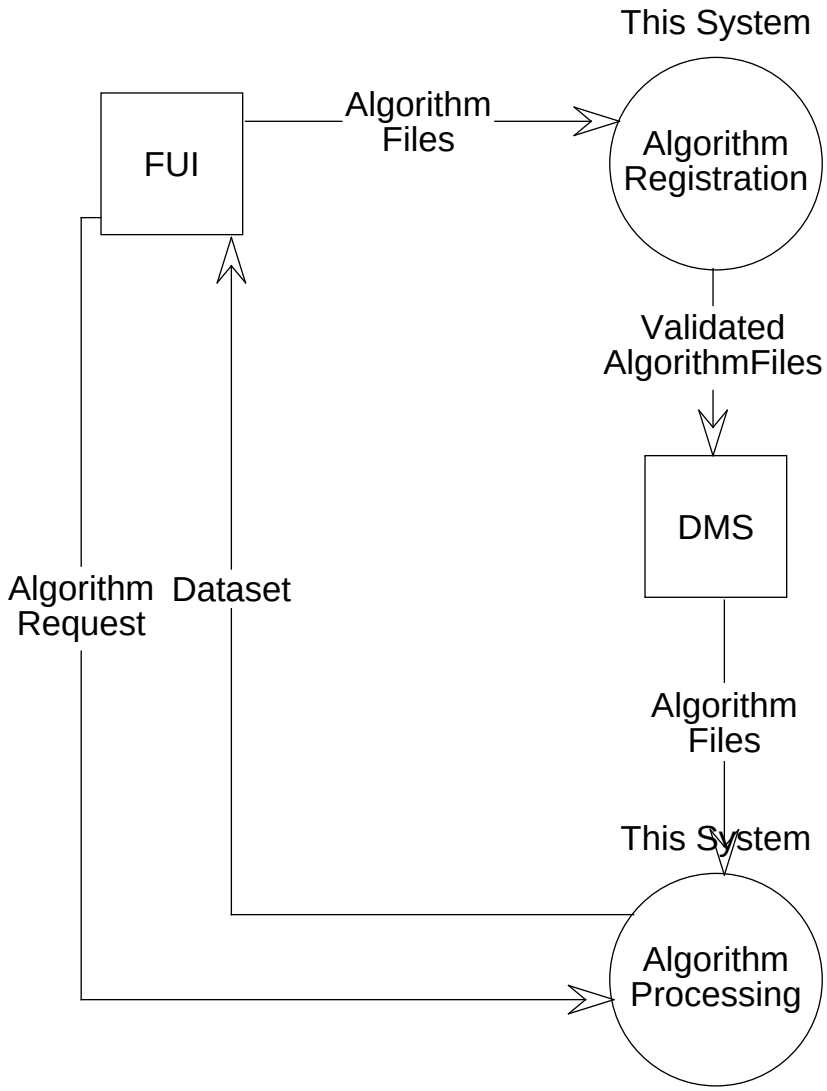

Figure 3.6-1. User Algorithms Context

3.6.2 User Algorithms Interfaces

Interface Service	Interface Class	Interface Class Description	Service Provider	Service User	Frequency
Provides Alogrithm Information	**FaDlAlgorith**m Registration	Contains algorithm name, shared object filename and path, and number of inputs/outputs	ANA	FUI	Whenever a new user algorithm is registered

3.6.3 User Algorithms Object Model

This model shows the relationship between the FaPaAlgorithmParamGroup, which controls the data flow, the FaDlUserAlgorithm, which controls the invocation of the algorithm, and the FaDlDyanmicLoader, which loads the user algorithm and invokes it. The FaDlDynamicLoader has multiple FaDlInputParameters and FaDlOutputParameters, corresponding to the number of inputs and outputs the User Algorithm has. Each FaDlInputParameter maps to a UserParameter, as well as the global variable in the User Algorithm shared object where the value of the parameter is expected to be. When the algorithm function is invoked, each FaDlInputParameter will load the value of it's UserParameter into the variable within the shared object associated with it. After the algorithm is invoked, the FaDlOutputParameters are checked to see if any output has been generated. If it has, then the output is placed in myAnalysisResults in the FaPaPaParameterGroup class. Also show is the algorithm registration class that is used to register the class.

- Refer to Figures 3.6-2 - 3.6-3 User Algorithms Object Model

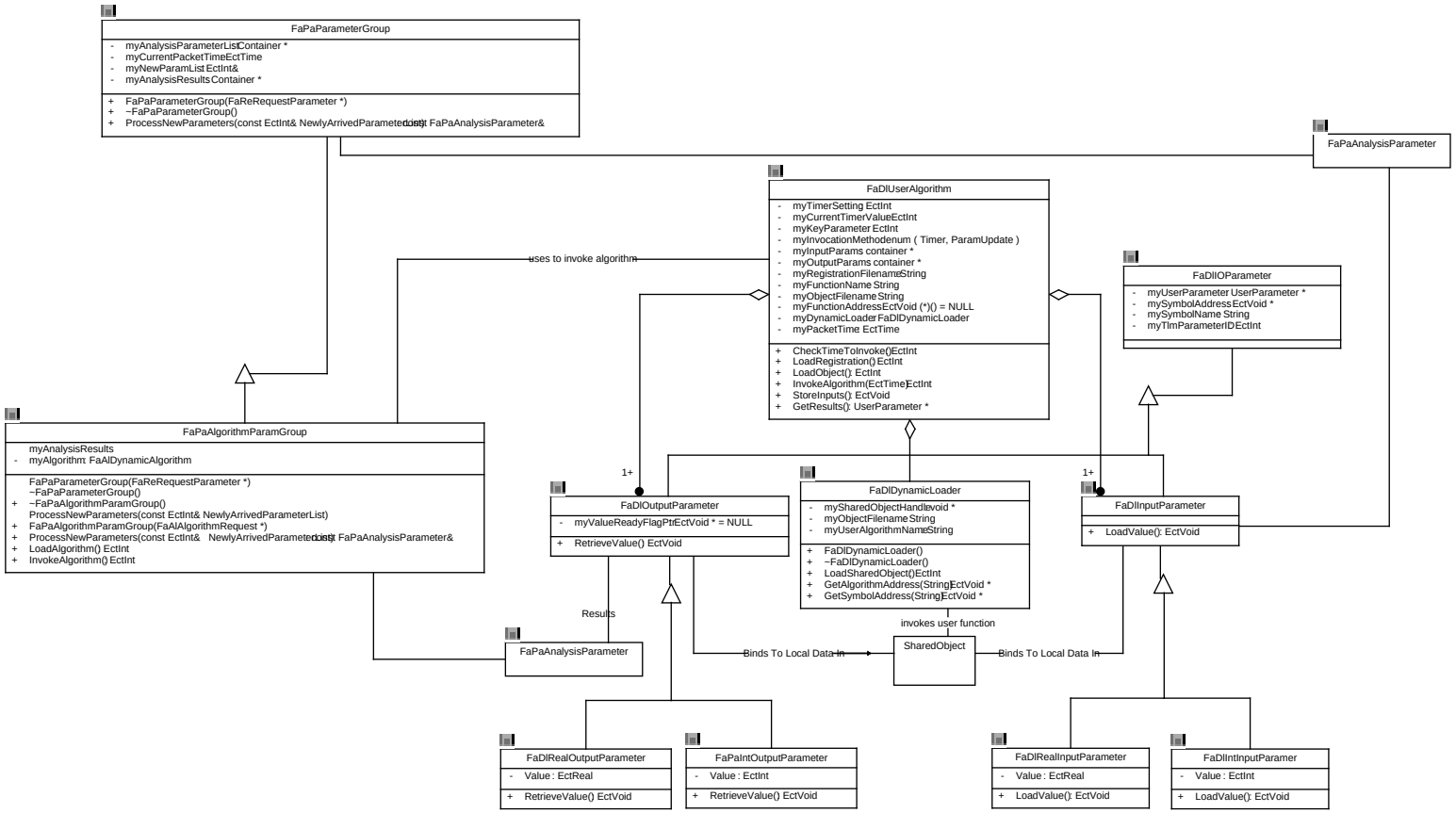

Figure 3.6-2. User Algorithms Object Model

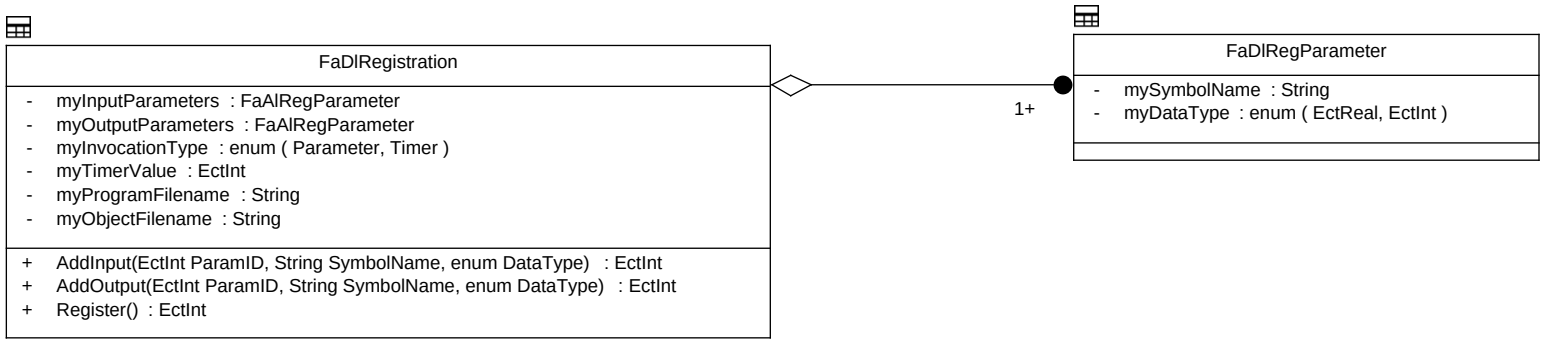

Figure 3.6-3. User Algorithms Object Model

3.6.4 User Algorithms Dynamic Model

3.6.4.1 User Algorithms Scenario Abstract

This scenario shows the flow of control for the application of a user defined algorithm on a time span of replay data.

Diagram 3.6-4 is the event trace for this scenario.

3.6.4.2 User Algorithms Summary Information

Interfaces:

None. Section 3.1 describes interfaces for an Analysis Request

Stimulus:

User Algorithms are performed as the result of user's Analysis Request being submitted. The Analysis Request contains a list of algorithms that are to be used to generate output to the dataset generated by the processing of the Analysis Request.

Desired Response:

The generation of output parameters from a user defined algorithm.

Pre-conditions:

The algorithm has been tested and registered with the EOC.

Post-conditions: None.

3.6.4.3 User Algorithms Scenario Description

- Refer to Figure 3.6-4. User Algorithms Event Trace

First, the FaDlUserAlgorithm loads the algorithm. When data begins to arrive, the FaPaAlgorithmParamGroup is informed of new data. Then, the FaDlUserAlgorithm is given a list of new parameters, and decides whether the algorithm should be invoked. If the algorithm is to be invoked, the FaDlDynamicLoader invokes the algorithm. After the algorithm is invoked, the FaDlUserAlgorithm retrieves the output values from the algorithm and forwards the results to the FaPaAlgorithmParamGroup, which in turn forwards the results to the FaPrDataset.

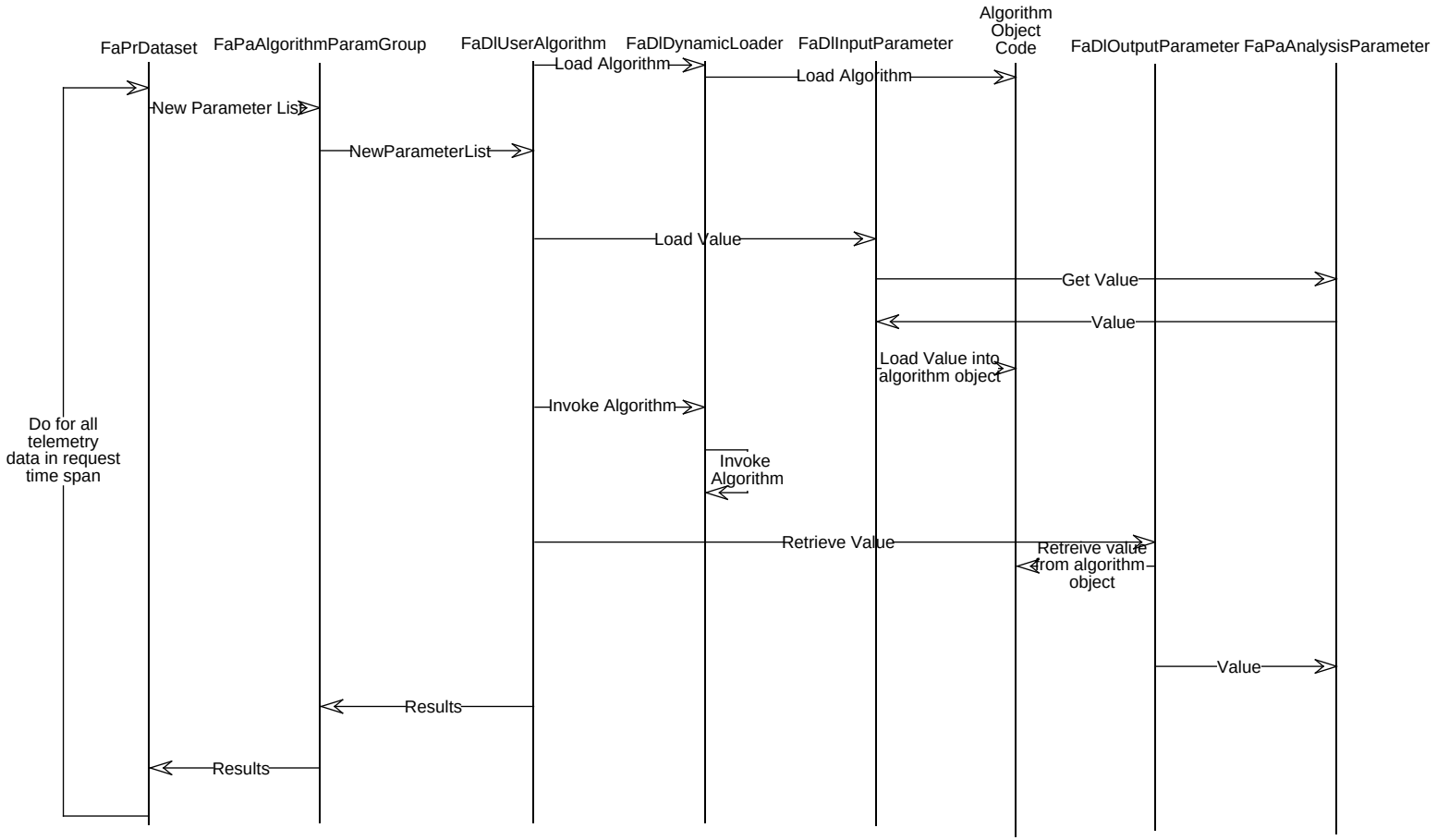

Figure 3.6-4. User Algorithms Event Trace

3.6.5 User Algorithms Data Dictionary

FaDlDynamicLoader

class **FaDlDynamicLoader**

This class is the tool which enables the user algorithm to be dynamically linked and loaded. The user algorithm will be in the form of shared object, which can be accessed by various library functions provided with the OS. In cases where the OS does not provide access to these types of objects at run-time, functions must be provided which allow similar access to the object file. This will hopefully be unnecessary, as this feature will hopefully become a POSIX standard soon. Even if it does not, many systems are making shared object access libraries standard. This class allows the user of the class to get the address of all available symbols within the object which has been dynamically linked.

Public Construction

FaDlDynamicLoader(void)
~FaDlDynamicLoader(void)

Public Functions

EctVoid* **GetAlgorithmAddress**(String)

This function will return the address of the function which is called when the algorithm is to be invoked. It will simply call GetSymbolAddress, but is included for clarity.

EctVoid* **GetSymbolAddress**(String)

When the shared object is loaded via the LoadSharedObject function, the addresses of all global symbols are made known. This function returns the address if the symbol represented by String

EctInt **LoadSharedObject**(void)

This Function loads the shared object from disk into memory. Sun versions will use dlopen (Filename) to load the shared object. HP versions will have a similar function call. Other platforms are TBD.

Private Data

String **myObjectFilename**

this is the name of the shared object file which is loaded from disk

void* **mySharedObjectHandle**

This is the pointer to the shared object information It should only be interpreted by functions which return information about the shared object. The only exception to this rule is that a NULL value indicates that no shared object is available.

String **myUserAlgorithmName**

The name of the function to be called when the algorithm is invoked

FaDlIOParameter

class **FaDlIOParameter**

This class is a base class for the input and output parameters used in invoking user algorithms. There are no operations associated with it, because operations are different for input and output parameters. There are identical attributes, however, hence the base class.

Each IOParameter has and associated User Parameter, complete with ParameterID, whether the IOParameter is a particular spacecraft parameter, or an algorithm output parameter. The IOParameter also maps to a symbol in the dynamically linked object, and attributes containing information about this association are provided.

Private Data

EctVoid* **mySymbolAddress**

This is the address of the symbol within the dynamically linked shared object.

String **mySymbolName**

This is the name of the symbol within the dynamically linked shared object which is associated with this IOParameter

`EctInt` **`myTlmParameterID`**

This is the ParameterID of the parameter associated with this IOParameter

`UserParameter*` **`myUserParameter`**

This is the address of the Userparameter which is associated with this IOParameter. The user parameter contains either the input value to be used in algorithm invocation, or the output value of one of the algorithm's output parameters.

FaDlInputParameter

`class` **`FaDlInputParameter`**

This is the base class for input parameters to dynamically linked algorithms. It inherits for FaDlIOParameter, and adds the operation LoadValue, which is virtual.

Base Classes

`public` **`FaDlIOParameter`**

Public Functions

`EctVoid` **`LoadValue`**`(void)`

This function will get the value of myUserParameter and load it into the symbol pointed to by mySymbolAddress It is a pure virtual function in this base class.

FaDlIntInputParamer

`class` **`FaDlIntInputParamer`**

class FaDlIntInputParameter

```
this class is used when the input symbol within
the dynamically linked algorithm is an integer
of type EctInt
```

Base Classes

`public` **`FaDlInputParameter`**

Public Functions

`EctVoid` **`RetrieveValue`**`(void)`

Function LoadValue

```
This function loads the value of myUserParameter
into the Symbol at address mySymbolAddress
The data type at mySymbolAddress is verified to be
an EctInt when the algorithm is registered
```

Private Data

`EctInt` **`Value`**

This attribute is a place holder for the value of myUserParameter

FaPaIntOutputParameter

`class` **`FaPaIntOutputParameter`**

class FaDlIntOutputParameter

```
this class is used when the output symbol within
the dynamically linked algorithm is an integer
of type EctInt
```

Base Classes

`public` **`FaDlOutputParameter`**

Public Functions

`EctVoid` **`RetrieveValue`**`(void)`

This function retrieves the value of the Symbol at address mySymbolAddress into myUserParameter The data type at mySymbolAddress is verified to be an EctInt when the algorithm is registered

Private Data

`EctInt` **`Value`**

This attribute is a place holder for the value of myUserParameter

FaDlOutputParameter

`class` **`FaDlOutputParameter`**

This is the base class for output parameters to dynamically linked algorithms. It inherits for FaDlIOParameter, and adds the operation RetrieveValue, which is virtual.

Base Classes

`public` **`FaDlIOParameter`**

Public Functions

`EctVoid` **`RetreiveValue`**`(void)`

Function RetrieveValue

```
This function will get the value of the symbol
pointed to by mySymbolAddress
and load it into myUserParameter
It is a pure virtual function in this base class.
```

FaDlRealInputParameter

`class` **`FaDlRealInputParameter`**

this class is used when the symbol within the dynamically linked algorithm is a real of type EctReal

Base Classes

`public` **`FaDlInputParameter`**

Public Functions

`EctVoid` **`LoadValue`**`(void)`

This function loads the value of myUserParameter into the Symbol at address mySymbolAddress The data type at mySymbolAddress is verified to be an EctInt when the algorithm is registered

Private Data

`EctReal` **`Value`**

This attribute is a place holder for the value of myUserParameter

FaDlRealOutputParameter

`class` **`FaDlRealOutputParameter`**

class FaDlIntOutputParameter

```
this class is used when the output symbol within
the dynamically linked algorithm is a real
of type EctReal
```

Base Classes

`public` **`FaDlOutputParameter`**

Public Functions

`EctVoid` **`RetrieveValue`**`(void)`

This function retrieves the value of the Symbol at address mySymbolAddress into myUserParameter The data type at mySymbolAddress is verified to be an EctReal when the algorithm is registered

Private Data

`EctReal` **`Value`**

This attribute is a place holder for the value of myUserParameter

FaDlUserAlgorithm

`class` **`FaDlUserAlgorithm`**

This class captures the high level functionality of the dynamically linked user defined algorithms it directs the invocation and loading of the algorithm.

Public Functions

`EctInt` **`CheckTimeToInvoke`**`(EctTime PacketTime)`

This function check the current packet time and decides if the algorithm should be invoked based on the length of time since the last invocation

`UserParameter*` **`GetResults`**`(void)`

This function retrieves resulting values from the user algorithm

`EctInt` **`InvokeAlgorithm`**`(EctTime)`

This function will cause the input parameters to be loaded into the appropriate shared object symbol addresses, and invokes the function pointed to by myFunctionAddress

`EctInt` **`LoadObject`**`(void)`

This function uses the FaDlDynamicLinker tool to load the object specified in the registration info

`EctInt` **`LoadRegistration`**`(void)`

This function loads the registration information and build the container of input and output params

`EctVoid` **`StoreInputs`**`(void)`

This is the function that is called to store the input values into the symbol addresses with the shared object

Private Functions

`enum(Timer, ParamUpdate)`

This variable determine whether the timer or key parameters are used in deciding when to invoke the algorithm.

Private Data

`EctInt` **`myCurrentTimerValue`**

This variable keeps track of the elapsed time since the last algorithm invocation if timer is used, or the current number of updates, if a key parameter is being used.

`FaDlDynamicLoader` **`myDynamicLoader`**

This is the tool that allows dynamic linking and loading of user algorithms. All OS dependent functionality is encapsulated here.

`String` **myFunctionName**

This is the name of the function to be called when the algorithm is invoked

`FaDlInputParameter*` **myInputParams**

container

`EctInt` **myKeyParameter**

myKeyParams

```
This is the parameter which should trigger
the algorithm's invocation upon it's update.
```

`String` **myObjectFilename**

This is the name of the object file containing the user algorithm.

`FaDlOutputParameter*` **myOutputParams**

container

`EctTime` **myPacketTime**

the current PacketTime

`String` **myRegistrationFilename**

This is the filename containing the registration information

`EctInt` **myTimerSetting**

This is length of time between algorithm invocations if timer is being used, or the number of updates needed by the key parameter, if a key parameter is being used

FaPaAlgorithmParamGroup

class **FaPaAlgorithmParamGroup**

stp/omt class definition 1432005

Base Classes

`public` **FaPaParameterGroup**

Public Construction

FaPaAlgorithmParamGroup`(FaAlAlgorithmRequest*)`
~FaPaAlgorithmParamGroup`(void)`

stp/omt class members

Public Functions

`EctInt` **InvokeAlgorithm**`(void)`
`EctInt` **LoadAlgorithm**`(void)`
`const FaPaAnalysisParameter&` **ProcessNewParameters**`(const EctInt& NewlyArrivedParameterList)`

Private Data

`FaAlDynamicAlgorithm` **myAlgorithm**

FaPaAnalysisParameter

class **FaPaAnalysisParameter**

stp/omt class definition 1280607

Base Classes

```
public RWCollectable
```

Public Functions

```
EctBoolean CheckDataQuality(void)
EctBoolean CheckIfCntReached(void)
EctBoolean CheckIfValueChanged(void)
EctVoid FaPaAnalysisParameter(EctVoid)
EctVoid RestoreGuts(void)
EctVoid SaveGuts(void)
UserParameter& UpdateParameter(void)
```

stp/omt class members

Private Data

```
EctInt myCnt
Raw myLastRawValue
EcTBoolean myOverrideFlag
EctInt myParameterId
EctInt mySampleRate
UserParameter* myUserParameterPtr
```

FaPaParameterGroup

class **FaPaParameterGroup**

stp/omt class definition 1280615

Public Construction

```
FaPaParameterGroup(FaReRequestParameter*)
~FaPaParameterGroup(void)
```

stp/omt class members

Public Functions

```
const FaPaAnalysisParameter& ProcessNewParameters(const EctInt&
      NewlyArrivedParameterList)
```

Private Data

```
Container* myAnalysisParameterList
Container* myAnalysisResults
EctTime myCurrentPacketTime
EctInt& myNewParamList
```

SharedObject

class **SharedObject**

stp/omt class definition 2194641

Abbreviations and Acronyms

ACL	Access Control List
AD	Acceptance Check/TC Data
AGS	ASTER Ground System
AM	Morning (ante meridian) -- see EOS AM
Ao	Availability
APID	Application Identifier
ARAM	Automated Reliability/Availability/Maintainability
ASTER	Advanced Spaceborne Thermal Emission and Reflection Radiometer (formerly ITIR)
ATC	Absolute Time Command
BAP	Baseline Activity Profile
BC	Bypass check/Control Commands
BD	Bypass check/TC Data (Expedited Service)
BDU	Bus Data Unit
bps	bits per second
CAC	Command Activity Controller
CCB	Change Control Board
CCSDS	Consultative Committee for Space Data Systems
CCTI	Control Center Technology Interchange
CD-ROM	Compact Disk-Read Only Memory
CDR	Critical Design Review
CDRL	Contract Data Requirements List
CERES	Clouds and Earth's Radiant Energy System
CI	Configuration item
CIL	Critical Items List
CLCW	Command Link Control Words
CLTU	Command Link Transmission Unit
CMD	Command subsystem
CMS	Command Management Subsystem
CODA	Customer Operations Data Accounting
COP	Command Operations Procedure
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit

CRC	Cyclic Redundancy Code
CSCI	Computer software configuration item
CSMS	Communications and Systems Management Segment
CSS	Communications Subsystem (CSMS)
CSTOL	Customer System Test and Operations Language
CTIU	Command and Telemetry Interface Unit (AM-1)
DAAC	Distributed Active Archive Center
DAR	Data Acquisition Request
DAS	Detailed Activity Schedule
DAT	Digital Audio Tape
DB	Data Base
DBA	Database Administrator
DBMS	Database Management System
DCE	Distributed Computing Environment
DCP	Default Configuration Procedure
DEC	Digital Equipment Corporation
DES	Data Encryption Standard
DFCD	Data Format Control Document
DID	Data Item Description
DMS	Data Management Subsystem
DOD	Digital Optical Data
DoD	Department of Defense
DS	Data Server
DSN	Deep Space Network
DSS	Decision Support System
e-mail	electronic mail
Ecom	EOS Communication
ECS	EOSDIS Core System
EDOS	EOS Data and Operations System
EDU	EDOS Data Unit
EGS	EOS Ground System
EOC	Earth Observation Center (Japan); EOS Operations Center (ECS)
EOD	Entering Orbital Day
EON	Entering Orbital Night
EOS	Earth Observing System

EOSDIS	EOS Data and Information System
EPS	Encapsulated Postscript
ESH	EDOS Service Header
ESN	EOSDIS Science Network
ETS	EOS Test System
EU	Engineering Unit
EUVE	Extreme Ultra Violet Explorer
FAS	FOS Analysis Subsystem
FAST	Fast Auroral Snapshot Explorer
FDDI	Fiber Distributed Data Interface
FDF	Flight Dynamics Facility
FDIR	Fault Detection and Isolation Recovery
FDM	FOS Data Management Subsystem
FMEA	Failure Modes and Effects Analyses
FOP	Frame Operations Procedure
FORMATS	FDF Orbital and Mission Aids Transformation System
FOS	Flight Operations Segment
FOT	Flight Operations Team
FOV	Field-Of-View
FPS	Fast Packet Switch
FRM	FOS Resource Management Subsystem
FSE	FOT S/C Evolutions
FTL	FOS Telemetry Subsystem
FUI	FOS User Interface
GB	Gigabytes
GCM	Global Circulation Model
GCMR	Global Circulation Model Request
GIMTACS	GOES I-M Telemetry and Command System
GMT	Greenwich Mean Time
GN	Ground Network
GOES	Geostationary Operational Environmental Satellite
GSFC	Goddard Space Flight Center
GUI	Graphical User Interface
H&S	Health and Safety
H/K	Housekeeking
HST	Hubble Space Telescope

I/F	Interface
I/O	Input/Output
ICC	Instrument Control Center
ICD	Interface Control Document
ID	Identifier
IDB	Instrument Database
IDR	Incremental Design Review
IEEE	Institute of Electrical and Electronics Engineers
IOT	Instrument Operations Team
IP	International Partners
IP-ICC	International Partners-Instrument Control Center
IPs	International Partners
IRD	Interface requirements document
ISDN	Integrated Systems Digital Network
ISOLAN	Isolated Local Area Network
ISR	Input Schedule Request
IST	Instrument Support Terminal
IST	Instrument Support Toolkit
IWG	Investigator Working Group
JPL	Jet Propulsion Laboratory
Kbps	Kilobits per second
LAN	Local Area Network
LaRC	Langley Research Center
LASP	Laboratory for Atmospheric Studies Project
LEO	Low Earth Orbit
LOS	Loss of Signal
LSM	Local System Manager
LTIP	Long-Term Instrument Plan
LTSP	Long-Term Science Plan
MAC	Medium Access Control; Message Authentication Code
MB	Megabytes
MBONE	Multicast Backbone
Mbps	Megabits per second
MDT	Mean Down Time
MIB	Management Information Base

MISR	Multi-angle Imaging Spectro-Radiometer
MMM	Minimum, Maximum, and Mean
MO&DSD	Mission Operations and Data Systems Directorate (GSFC Code 500)
MODIS	Moderate resolution Imaging Spectrometer
MOPITT	Measurements Of Pollution In The Troposphere
MSS	Management Subsystem
MTPE	Mission to Planet Earth
NASA	National Aeronautics and Space Administration
Nascom	NASA Communications Network
NASDA	National Space Development Agency (Japan)
NCAR	National Center for Atmospheric Research
NCC	Network Control Center
NEC	North Equator Crossing
NFS	Network File System
NOAA	National Oceanic and Atmospheric Administration
NSI	NASA Science Internet
NTT	Nippon Telephone and Telegraph
OASIS	Operations and Science Instrument Support
ODB	Operational Database
ODM	Operational Data Message
OMT	Object Model Technique
OO	Object Oriented
OOD	Object Oriented Design
OpLAN	Operational LAN
OSI	Open System Interconnect
PACS	Polar Acquisition and Command System
PAS	Planning and Scheduling
PDB	Project Data Base
PDF	Publisher's Display Format
PDL	Program Design Language
PDR	Preliminary Design Review
PI	Principal Investigator
PI/TL	Principal Investigator/Team Leader
PID	Parameter ID
PIN	Password Identification Number
POLAR	Polar Plasma Laboratory

POP	Polar-Orbiting Platform
POSIX	Portable Operating System for Computing Environments
PSAT	Predicted Site Acquisition Table
PSTOL	PORTS System Test and Operation Language
Q/L	Quick Look
R/T	Real-Time
RAID	Redundant Array of Inexpensive Disks
RCM	Real-Time Contact Management
RDBMS	Relational Database Management System
RMA	Reliability, Maintainability, Availability
RMON	Remote Monitoring
RMS	Resource Management Subsystem
RPC	Remote Processing Computer
RTCS	Relative Time Command Sequence
RTS	Relative Time Sequence; Real-Time Server
S/C	Spacecraft
SAR	Schedule Add Requests
SCC	Spacecraft Controls Computer
SCF	Science Computing Facility
SCL	Spacecraft Command Language
SDF	Software Development Facility
SDPS	Science Data Processing Segment
SDVF	Software Development and Validation Facility
SEAS	Systems, Engineering, and Analysis Support
SEC	South Equator Crossing
SLAN	Support LAN
SMA	S-band Multiple Access
SMC	Service Management Center
SN	Space Network
SNMP	System Network Mgt Protocol
SQL	Structured Query Language
SSA	S-band Single Access
SSIM	Spacecraft Simulator
SSR	Solid State Recorder
STOL	System Test and Operations Language

T&C	Telemetry and Command
TAE	Transportable Applications Environment
TBD	To Be Determined
TBR	To Be Replaced/Resolved/Reviewed
TCP	Transmission Control Protocol
TD	Target Day
TDM	Time Division Multiplex
TDRS	Tracking and Data Relay Satellite
TDRSS	Tracking and Data Relay Satellite System
TIROS	Television Infrared Operational Satellite
TL	Team Leader
TLM	Telemetry subsystem
TMON	Telemetry Monitor
TOO	Target Of Opportunity
TOPEX	Topography Ocean Experiment
TPOCC	Transportable Payload Operations Control Center
TRMM	Tropical Rainfall Measuring Mission
TRUST	TDRSS Resource User Support Terminal
TSS	TDRSS Service Session
TSTOL	TRMM System Test and Operations Language
TW	Target Week
U.S.	United States
UAV	User Antenna View
UI	User Interface
UPS	User Planning System
US	User Station
UTC	Universal Time Code; Universal Time Coordinated
VAX	Virtual Extended Address
VMS	Virtual Memory System
W/S	Workstation
WAN	Wide Area Network
WOTS	Wallops Orbital Tracking Station
XTE	X-Ray Timing Explorer

This page intentionally left blank.

Glossary

GLOSSARY of TERMS for the Flight Operations Segment

activity	A specified amount of scheduled work that has a defined start date, takes a specific amount of time to complete, and comprises definable tasks.
analysis	Technical or mathematical evaluation based on calculation, interpolation, or other analytical methods. Analysis involves the processing of accumulated data obtained from other verification methods.
attitude data	Data that represent spacecraft orientation and onboard pointing information. Attitude data includes: o Attitude sensor data used to determine the pointing of the spacecraft axes, calibration and alignment data, Euler angles or quaternions, rates and biases, and associated parameters. o Attitude generated onboard in quaternion or Euler angle form. o Refined and routine production data related to the accuracy or knowledge of the attitude.
availability	A measure of the degree to which an item is in an operable and committable state at the start of a "mission" (a requirement to perform its function) when the "mission" is called for an unknown (random) time. (Mathematically, operational availability is defined as the mean time between failures divided by the sum of the mean time between failures and the mean down time [before restoration of function].

availability (inherent) (A_i)

The probability that, when under stated conditions in an ideal support environment without consideration for preventive action, a system will operate satisfactorily at any time. The "ideal support environment" referred to, exists when the stipulated tools, parts, skilled work force manuals, support equipment and other support items required are available. Inherent availability excludes whatever ready time, preventive maintenance downtime, supply downtime and administrative downtime may require. A_i can be expressed by the following formula:

$$A_i = MTBF / (MTBF + MTTR)$$

Where: MTBF = Mean Time Between Failures
MTTR = Mean Time To Repair

availability (operational) (A_o)

The probability that a system or equipment, when used under stated conditions in an actual operational environment, will operate satisfactorily when called upon. A_o can be expressed by the following formula:

$$A_o = MTBM / (MTBM + MDT + ST)$$

Where: MTBM = Mean Time Between Maintenance (either corrective or preventive)
MDT = Mean Maintenance Down Time where corrective, preventive administrative and logistics actions are all considered.
ST = Standby Time (or switch over time)

A schedule of activities for a target week corresponding to normal instrument operations constructed by integrating long term plans (i.e., LTSP, LTIP, and long term spacecraft operations plan).

build

An assemblage of threads to produce a gradual buildup of system capabilities.

calibration

The collection of data required to perform calibration of the instrument science data, instrument engineering data, and the spacecraft engineering data. It includes pre-flight calibration measurements, in-flight calibrator measurements, calibration equation coefficients derived from calibration software routines, and ground truth data that are to be used in the data calibration processing routine.

command	Instruction for action to be carried out by a space-based instrument or spacecraft.
command and data handling (C&DH)	The spacecraft command and data handling subsystem which conveys commands to the spacecraft and research instruments, collects and formats spacecraft and instrument data, generates time and frequency references for subsystems and instruments, and collects and distributes ancillary data.
command group	A logical set of one or more commands which are not stored onboard the spacecraft and instruments for delayed execution, but are executed immediately upon reaching their destination on board. For the U.S. spacecraft, from the perspective of the EOS Operations Center (EOC), a preplanned command group is preprocessed by, and stored at, the EOC in preparation for later uplink. A real-time command group is unplanned in the sense that it is not preprocessed and stored by the EOC.
detailed activity schedules	The schedule for a spacecraft and instruments which covers up to a10 day period and is generated/updated daily based on the instrument activity listing for each of the instruments on the respective spacecraft. For a spacecraft and instrument schedule the spacecraft subsystem activity specifications needed for routine spacecraft maintenance and/or for supporting instruments activities are incorporated in the detailed activity schedule.
direct broadcast	Continuous down-link transmission of selected real-time data over a broad area (non-specific users).

EOS Data and Operations System (EDOS) production data set	Data sets generated by EDOS using raw instrument or spacecraft packets with space-to-ground transmission artifacts removed, in time order, with duplicate data removed, and with quality/ accounting (Q/A) metadata appended. Time span or number of packets encompassed in a single data set are specified by the recipient of the data. These data sets are equivalent to Level 0 data formatted with Q/A metadata. For EOS, the data sets are composed of: instrument science packets, instrument engineering packets, spacecraft housekeeping packets, or onboard ancillary packets with quality and accounting information from each individual packet and the data set itself and with essential formatting information for unambiguous identification and subsequent processing.
housekeeping data	The subset of engineering data required for mission and science operations. These include health and safety, ephemeris, and other required environmental parameters.
instrument	o A hardware system that collects scientific or operational data. o Hardware-integrated collection of one or more sensors contributing data of one type to an investigation. o An integrated collection of hardware containing one or more sensors and associated controls designed to produce data on/in an observational environment.
instrument activity deviation list	An instrument's activity deviations from an existinginstrument activity list, used by the EOC for developing the detailed activity schedule.
instrument activity list	An instrument's list of activities that nominally covers seven days, used by the EOC for developing the detailed activity schedule.
instrument engineering data	Subset of telemetered engineering data required for performing instrument operations and science processing
instrument microprocessor memory loads	Storage of data into the contents of the memory of an instrument’s microprocessor, if applicable. These loads could include microprocessor-stored tables, microprocessor-stored commands, or updates to microprocessor software.

instrument resource deviation list	An instrument's anticipated resource deviations from anexisting resource profile, used by the EOC for establishing TDRSS contact times and building the preliminary resource schedule.
instrument resource profile	Anticipated resource needs for an instrument over a targetweek, used by the EOC for establishing TDRSS contact times and building the preliminary resource schedule.
instrument science data	Data produced by the science sensor(s) of an instrument, usually constituting the mission of that instrument.
long-term instrument plan (LTIP)	The plan generated by the instrument representative to the spacecraft's IWG with instrument-specific information to complement the LTSP. It is generated or updated approximately every six months and covers a period of up to approximately 5 years.
long-term science plan (LTSP)	The plan generated by the spacecraft's IWG containing guidelines, policy, and priorities for its spacecraft and instruments. The LTSP is generated or updated approximately every six months and covers a period of up to approximately five years.
long term spacecraft operations plan	Outlines anticipated spacecraft subsystem operations and maintenance, along with forecasted orbit maneuvers from the Flight Dynamics Facility, spanning a period of several months.
mean time between failure (MTBF)	The reliability result of the reciprocal of a failure rate that predicts the average number of hours that an item, assembly or piece part will operate within specific design parameters. (MTBF=1/(l) failure rate; (l) failure rate = # of failures/ operating time.
mean down time (MDT)	Sum of the mean time to repair MTTR plus the average logistic delay times.
mean time between maintenance (MTBM)	The mean time between preventive maintenance (MTBPM) and mean time between corrective maintenance (MTBCM) of the ECS equipment. Each will contribute to the calculation of the MTBM and follow the relationship: 1/MTBM = 1/MTBPM + 1/MTBCM
mean time to repair (MTTR)	The mean time required to perform corrective maintenance to restore a system/equipment to operate within design parameters.

object	Identifiable encapsulated entities providing one or more services that clients can request. Objects are created and destroyed as a result of object requests. Objects are identified by client via unique reference.
orbit data	Data that represent spacecraft locations. Orbit (or ephemeris) data include: Geodetic latitude, longitude and height above an adopted reference ellipsoid (or distance from the center of mass of the Earth); a corresponding statement about the accuracy of the position and the corresponding time of the position (including the time system); some accuracy requirements may be hundreds of meters while other may be a few centimeters.
playback data	Data that have been stored on-board the spacecraft for delayed transmission to the ground.
preliminary resource schedule	An initial integrated spacecraft schedule, derived from instrument and subsystem resource needs, that includes the network control center TDRSS contact times and nominally spans seven days.
preplanned stored command	A command issued to an instrument or subsystem to be executed at some later time. These commands will be collected and forwarded during an available uplink prior to execution.
principal investigator (PI)	An individual who is contracted to conduct a specific scientific investigation. (An instrument PI is the person designated by the EOS Program as ultimately responsible for the delivery and performance of standard products derived from an EOS instrument investigation.)
prototype	Prototypes are focused developments of some aspect of the system which may advance evolutionary change. Prototypes may be developed without anticipation of the resulting software being directly included in a formal release. Prototypes are developed on a faster time scale than the incremental and formal development track.

raw data	Data in their original packets, as received from the spacecraft and instruments, unprocessed by EDOS. o Level 0 – Raw instrument data at original resolution, time ordered, with duplicate packets removed. o Level 1A – Level 0 data, which may have been reformatted or transformed reversibly, located to a coordinate system, and packaged with needed ancillary and engineering data. o Level 1B – Radiometrically corrected and calibrated data in physical units at full instrument resolution as acquired. o Level 2 – Retrieved environmental variables (e.g., ocean wave height, soil moisture, ice concentration) at the same location and similar resolution as the Level 1 source data. o Level 3 – Data or retrieved environmental variables that have have been spatially and/or temporally resampled (i.e., derived from Level 1 or Level 2 data products). Such resampling may include averaging and compositing. o Level 4 – Model output and/or variables derived from lower level data which are not directly measured by the instruments. For example, new variables based upon a time series of Level 2 or Level 3 data.
real-time data	Data that are acquired and transmitted immediately to the ground (as opposed to playback data). Delay is limited to the actual time required to transmit the data.
reconfiguration	A change in operational hardware, software, data bases or procedures brought about by a change in a system's objectives.
SCC-stored commands and tables	Commands and tables which are stored in the memory of the central onboard computer on the spacecraft. The execution of these commands or the result of loading these operational tables occurs sometime following their storage. The term "core-stored" applies only to the location where the items are stored on the spacecraft and instruments; core-stored commands or tables could be associated with the spacecraft or any of the instruments.
scenario	A description of the operation of the system in user's terminology including a description of the output response for a given set of input stimuli. Scenarios are used to define operations concepts.

segment	One of the three functional subdivisions of the ECS: CSMS -- Communications and Systems Management Segment FOS -- Flight Operations Segment SDPS -- Science Data Processing Segment
sensor	A device which transmits an output signal in response to a physical input stimulus (such as radiance, sound, etc.). Science and engineering sensors are distinguished according to the stimuli to which they respond. o Sensor name: The name of the satellite sensor which was used to obtain that data.
spacecraft engineering data	The subset of engineering data from spacecraft sensor measurements and on-board computations.
spacecraft subsystems activity list	A spacecraft subsystem's list of activities that nominally covers seven days, used by the EOC for developing the detailed activity schedule.
spacecraft subsystems resource profile	Anticipated resource needs for a spacecraft subsystem over a target week, used by the EOC for establishing TDRSS contact times and building the preliminary resource schedule.
target of opportunity (TOO)	A TOO is a science event or phenomenon that cannot be fully predicted in advance, thus requiring timely system response or high-priority processing.
thread	A set of components (software, hardware, and data) and operational procedures that implement a function or set of functions.
thread, *as used in some Systems Engineering documents*	A set of components (software, hardware, and data) and operational procedures that implement a scenario, portion of a scenario, or multiple scenarios.
toolkits	Some user toolkits developed by the ECS contractor will be packaged and delivered on a schedule independent of ECS releases to facilitate science data processing software development and other development activities occurring in parallel with the ECS.